Computers and Mathematics with Applications 58 (2009) 1071–1080

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

# Towards a hybrid parallelization of lattice Boltzmann methods

Vincent Heuveline [a], Mathias J. Krause [a,*], Jonas Latt [b]

[a] Universität Karlsruhe (TH), Fritz-Erler-Str. 23, 76133 Karlsruhe, Germany

[b] Ecole Polytechnique Federale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland

ARTICLE INFO

*Keywords:*
Lattice Boltzmann method
CFD
High performance computing
MPI
OpenMP
Parallelization

ABSTRACT

Ongoing research towards the development of a hybrid parallelization concept for lattice Boltzmann methods is presented. It allows coping with platforms sharing both the properties of shared and distributed architectures. The proposed approach relies on spatial domain decomposition where each domain represents a basic block entity which is solved on a symmetric multi-processing (SMP) system. Emphasis is placed on the software design and the reworking needed to achieve good performance using OpenMP in that context. Those ideas are implemented in the C++ project OpenLB, which is also sketched in this article. The efficiency of the proposed approaches is tested on a 3D benchmark problem and compared with a purely MPI based approach.



## 1. Introduction

In the last decade lattice Boltzmann methods (LBM) have evolved into a mature tool in computational fluid dynamics (CFD) and related topics in the landscape of both commercial and academic softwares. 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. In that framework it is however a common pitfall to underestimate the complexity of the associated schemes needed to obtain the high performance provided by contemporary computing architectures. New trends in computer architectures such as the multi-core CPUs and coprocessor technologies require specific changes with respect to the mathematical model, algorithm setup, software design and implementation strategies.

Currently there are intensive research efforts to analyze, develop and adapt adequate lattice Boltzmann approaches for dedicated hardware architectures. Present typical examples of this trend are developments related to IBM Cell processors, Graphic Processing Units, Clearspeed accelerator board and multi-core processors from AMD, Intel and others. These new technologies blur the line of separation between architectures with shared and distributed memory. In that context the authors are convinced that the development of efficient hybrid parallelization schemes for LBM does not only represent a major challenge in the near future but may become a sine qua non condition to take advantage of the performance both on high performance computing (HPC) hardware and on *commodity off the shelf* (COTS) hardware.

The goal of this paper is to present ongoing research towards the development of a generic hybrid parallelization concept for LBM allowing coping with platforms sharing both the properties of shared and distributed architecture platforms. The proposed approach relies on a spatial domain decomposition where each domain represents a basic block entity which is solved on a SMP system. The realization of the proposed concept is illustrated in the framework of the C++ project OpenLB. Besides the model and conceptional aspects, emphasis of this report is placed on the software design and the reworking needed to achieve good performance using OpenMP in that context. It is important to note that currently no standard exists

* Corresponding author.
*E-mail address:* mathias.krause@kit.edu (M.J. Krause).

0898-1221/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2009.04.001

for an adequate software design on multi-core and coprocessor based platforms. In this paper our aim is also to address and reveal peculiarities of the implementation of LBM for such platforms.

The remainder of this article is organized as follows. Section 2 is dedicated to the formulation of prototypical LB algorithms. In Section 3 we present the proposed hybrid parallelization concept and its realization in the OpenLB code. In Section 4 we address specific issues related to the parallelization of lattice blocks on SMP using OpenMP. In Section 5 numerical experiments and benchmark results are presented. The efficiencies of the implementations are finally tested on a 3D benchmark problem and compared with another approach using MPI as parallelization paradigm.

## 2. Lattice Boltzmann methods (LBM)

A lattice Boltzmann numerical model simulates the dynamics of particle distribution functions in phase space. These techniques find their application, among others, in the simulation of the kinetic equations for fluids. An exhaustive derivation of the LB equations can be found in the literature [1–3].

The distribution functions are governed by the Boltzmann equation, a balance equation between particle transport and collisions:

$$\partial_t f + \vec{v} \cdot \vec{\nabla}_{\vec{r}} f = \Omega(f), \tag{1}$$

where $f = f(\vec{r}, \vec{v}, t)$ is defined in the phase space with position $\vec{r}$ and velocity $\vec{v}$ at the time $t$.

In the BGK model, the fluid is taken to be close to its local equilibrium, and its dynamics is governed by a relaxation towards this equilibrium:

$$\Omega(f) = \Omega(\vec{r}, t) = -\omega(f(\vec{r}, \vec{v}, t) - f^{eq}(\rho(\vec{r}, t), \vec{u}(\vec{r}, t))), \tag{2}$$

where $\omega$ is a fluid specific relaxation frequency.

In numerical LBM, the continuous transient phase space is replaced by a discrete space with a spacing $\delta r$ for the positions, a set of $q$ vectors $\vec{c}_i$ for the velocities and a spacing $\delta t$ for time. To begin with, the continuous space of positions $\vec{r}$ is represented by a discrete set of points, which in 2D we label as $\vec{r}_{xy}$ by two indexes, and in 3D as $\vec{r}_{xyz}$ by three indexes. Those points are displayed on a regular grid with the same constant spacing $\delta r$ in all space directions: $\vec{r}_{xy} = \vec{r}_0 + (x\,\delta r, y\,\delta r)$ and $\vec{r}_{xyz} = \vec{r}_0 + (x\,\delta r, y\,\delta r, z\,\delta r)$. The space of velocities is similarly represented by a discrete set of $q$ vectors $\vec{c}_i$, $i = 0 \cdots q-1$. They are chosen in such a way that every vector $\vec{c}_i$ connects a grid point $\vec{r}_{xy}$ at a time $t$ with some other grid point $\vec{r}_{x'y'}$ at the next time step $t + \delta t$ : $\vec{r}_{x'y'}(t+1) = \vec{r}_{xy}(t) + \vec{c}_i$. The resulting discrete phase space is called the lattice and is labeled by the term DdQq by the numbers of space dimensions $d$ and the number of discrete velocities $q$. To reflect the discretization of velocity space, the continuous distribution function $f$ is replaced by a set of $q$ distribution functions $f_i$, representing an average value of $f$ in the vicinity of the velocity $\vec{c}_i$.

Assuming adequate scaling, the iterative process to be solved in the LB algorithm is written as follows:

$$f_i(\vec{r} + \vec{c}_i, t+1) - f_i(\vec{r}, t) = -\omega\left(f_i(\vec{r}, t) - f_i^{eq}\left(\rho(\vec{r}, t), \vec{u}(\vec{r}, t)\right)\right) \quad \text{for } i = 0 \cdots q-1, \tag{3}$$

where

$$f_i^{eq}(\rho, \vec{u}) = \rho t_i \left(1 + 3\,\vec{c}_i \cdot \vec{u} + \frac{9}{2}\left|\vec{c}_i \cdot \vec{u}\right|^2 - \frac{3}{2}\left|\vec{u}\right|^2\right), \tag{4}$$

$$\rho = \sum_{i=0}^{q-1} f_i \quad \text{and} \quad \rho\vec{u} = \sum_{i=0}^{q-1} \vec{c}_i f_i. \tag{5}$$

The $t_i$ and $\vec{c}_i$ are lattice-dependent constants. This LB framework, based on a BGK approximation of the collision, has been consistently discussed in [4]. It is shown that the LB dynamics is asymptotically equivalent to the dynamics of the Navier–Stokes equations and that the fluid viscosity $\nu$ is directly related to the relaxation parameter $\omega$. This equivalence is reached in the limit of small grid spacing and time steps and in a regime of small Mach and Knudsen numbers.

## 3. Hybrid parallelization in OpenLB

### 3.1. The OpenLB project

OpenLB is an effort to set up a lattice Boltzmann code for the simulation of fluid flows. The code is intended to be used both by application programmers who simply want to run a simulation with a given flow geometry, and by developers who implement their own particular dynamics. A main goal with respect to the design of the OpenLB code is to obtain a straightforward and intuitive implementation of LB models with almost no loss of efficiency. In the design of OpenLB a main emphasis is given to genericity in its many facets. Generic programming allows to develop a single code that can serve many purposes. The full code is written in C++ and takes advantage of both the dynamic and static genericity.

The OpenLB source code is publicly available on the project web site [5]. It is cross-verified for software quality by several reviewers and is presented along with a user guide. To the knowledge of the authors, the OpenLB project is the

first attempt to produce a generic platform for LB programming and to share it with the community via a system of open source contributions. However, implementation details and performance issues related to the LBM have been discussed previously in the literature, as e.g. in [6–9], and it has also been suggested previously to use object-oriented techniques for the implementation of LB code [10].

### 3.2. Organization of the code

The core of OpenLB consists of a `BlockLattice`, a simple and efficient array-like construct. This object executes a lattice Boltzmann algorithm in a very traditional sense as indicated in Algorithm 1. All `Cells` of the `BlockLattice` are iteratively parsed, and a local collision step is executed, followed by a non-local streaming step. The streaming step is independent of the choice of lattice Boltzmann dynamics and remains invariant. The collision step on the other hand determines the physics of the model and can be configured by the user, by attributing a fully configurable dynamics object to each `Cell`. In this way, it is easy to implement inhomogeneous fluids which use a different type of physics from one `Cell` to another.

Although this concept of a `BlockLattice` is neat and should please the programmer by being conceptually close to the theory of the lattice Boltzmann method, it is not sufficiently general to address all possible issues arising in real life. As a case in point, some boundary conditions are non-local and need to access neighbouring nodes. Their implementation does therefore not fit into the framework of a `BlockLattice` exposed previously. The philosophy of OpenLB takes for granted that such situations, although they arise, take place in spatially confined areas only, as for example the domain boundaries. They may therefore be implemented by slightly less efficient means, without spoiling the overall efficiency of the code. Their execution is taken care of by a postprocessing step (see Algorithm 1), which, instead of stepping over the whole lattice a second time, parses selected `Cells` only.

**Algorithm 1** Basic lattice Boltzmann algorithm

1. reading input
2. simulation setup
3. time loop

**for** $t = 0$ to $t = t_{max}$ **do**

a) collision
b) streaming
c) postprocessing
d) writing output

Another task that falls under the responsibility of the postprocessing step is the interconnection of various `BlockLattice` structures during the implementation of higher level constructs described in the next section. In this way, the efficiency of a basic lattice Boltzmann method is combined with an access to advanced constructs, including parallel and grid refined codes.

### 3.3. Data structure design

The lattice Boltzmann method, in its most widely accepted formulation, is executed on a regular, homogeneous lattice with equal grid spacing in all directions. When numerical constraints require that a given problem is solved on an inhomogeneous grid, it is common to adopt a so-called *multi-block* approach: the computational domain is partitioned into subgrids with different levels of resolution, and the interface between those subgrids is handled appropriately. This approach appears to respect the spirit of LB method well and leads to implementations that are both elegant and efficient. Furthermore, it encourages a particularly efficient form of *data parallelism*, in which an array is cut into regular pieces and distributed over the nodes of a parallel machine. As a result, LB applications can be run even on large parallel machines with a particularly satisfying gain of speed.

The same spirit is adopted in the OpenLB package, in which the basic datastructure is a `BlockLattice` which represents a regular array of `Cells`. In each `Cell` the $q$ variables for the storage of the discrete velocity distribution functions are contiguous in memory (cf. collision optimized data layout in [11]). Required memory is allocated only once, as no temporary memory is needed in the applied algorithm (see Section 4.3). This data structure on its turn 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 (see Fig. 1).

### 3.4. Parallel implementation details

The most time demanding steps in lattice Boltzmann simulations are the collision and streaming. Even for applications with complex geometries the execution of these two program parts will take easily more than 95% of the total execution

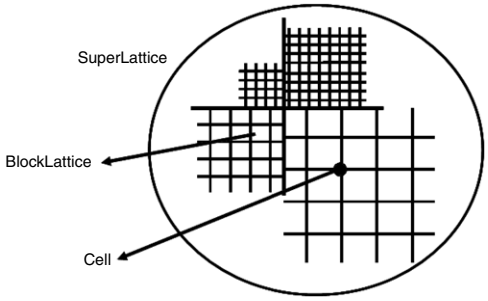


**Fig. 1.** Data structure in OpenLB: A number of `BlockLattices` build a `SuperLattice` to adopt higher level software constructs like multi-block, grid refined lattices and parallelized lattices.

time if the lattice size is $100^3$ and more. Since the collision step is purely local and the streaming step only requires data of the neighbouring nodes parallelizing by domain decomposition is efficient since the communication costs are low. This is shown for example in [12–16]. Following this classical approach it is not categorically necessary to change the basic structure of the sequential lattice Boltzmann algorithm (see Algorithm 1). The usage of pre-processor programming enables to write one single code for the sequential and several parallel modes. The few parts of the program that require special treatment for the parallel case can be handled by pre-processor directives. An advantage is that one single code simplifies the development of new code. Moreover there is no loss in efficiency since always either the sequential or a parallel code is compiled. In combination with the usage of a modern object-oriented and template based programming language, the adoption of several parallelization paradigms like OpenMP and MPI and finally their combination is possible.

To realize a hybrid parallelization in OpenLB, MPI is used for communication between the `BlockLattices` within a `SuperLattice` and OpenMP to parallelize each single `BlockLattice`. This approach mirrors the architecture of modern high performance computers as they are described in Section 4.2. Pre-processor programming is introduced by setting a variable `PARALLEL_MODE` in the makefile to either `OFF`, `MPI`, `OMP` or `HYBRID`. Another variable is passed to the compiler and the pre-processor can then prepare the corresponding source code. To simplify the implementation a MPI and OpenMP manager is introduced. Needed methods of the corresponding libraries as well as global static settings are provided through the managers. A global instance is given to enable access to subroutines.

## 4. OpenMP based parallelization

### *4.1. OpenMP specific restrictions*

OpenMP (Open Multi-Processing) is an application programming interface for portable shared memory multi-processing programming (see e.g. [17] and the references therein). The main design goal is to keep the sequential functionality and the source code structure of the program while providing flexible but simple ways to annotate the program that specific parts can be run in parallel and this meets the requirements for a hybrid parallelization concept as is introduced in Section 3.4.

OpenMP is characterized by a rather simple semantic which makes it difficult to capture the complexity of the underlying hardware leading to the following possibly critical issues:

- **Synchronization overhead:** Frequent change of control flow between one sequential main task and parallel parts can reduce the speedup. Even though there is a huge number of time steps and the execution time for one time step is rather small this is not a severe problem. The EPCC OpenMP Microbenchmarks [18] show an overhead per loop construct of about 5 $\mu$s, with up to 25 $\mu$s when using a reduction operation. This is only tolerable considering applications of adequate lattice size where the execution of one time step takes many times over these overhead times. For a realistic problem with a lattice size of $100^3$ for example the execution takes 0.53 s (1.9 MLUPS[1]) on an AMD Opteron with clock speed of 2.6 GHz on *HP XC4000* (see 4.2 for specifications).
- **Shared access to common data:** Due to the very regular data structures which allowed domain decomposition this is not an issue in OpenLB.
- **Shared memory bandwidth:** This is the most severe problem because the underlying data structure is larger than the provided cache. The resulting bottleneck can be reduced by fusing the two loops for the collision and streaming step together.

The OpenMP programming paradigm supports shared memory parallel programming on many architectures. However, it does not consider the hierarchy of the memory system i.e. it assumes uniform memory access (UMA) where the times to access specific parts of the memory are assumed to be equal. This is in contrast to the actual hardware of shared memory

[1] Million fluid-lattice-site updates per second, here for a D3Q19 BGK model.

```
template<typename T, template<typename U> class Lattice>
void BlockLattice2D<T,Lattice>::
                  collide(int x0, int x1, int y0, int y1) {
  OLB_PRECONDITION(x0>=0 && x1<nx);
  OLB_PRECONDITION(x1>=x0);
  OLB_PRECONDITION(y0>=0 && y1<ny);
  OLB_PRECONDITION(y1>=y0);

  int iX, iY;
  #ifdef PARALLEL_MODE_OMP
  #pragma omp parallel for private (iY) schedule(dynamic,1)
  #endif
  for (iX=x0; iX<=x1; ++iX) {
    for (iY=y0; iY<=y1; ++iY) {
      grid[iX][iY].collide(getStatistics());
      grid[iX][iY].revert();
    }
  }
}
```

**Fig. 2.** The OpenLB source code for a parallelized collision step (c).

compute nodes. They often have non-uniform memory access (NUMA), the memory is partitioned into different parts which are local to a specific processor. Unfortunately these different partitions are not considered in current programming environments, where one cannot specify to which processor the result of a memory allocation should be local. Even worse, because of the shared memory paradigm, one usually has only one memory allocation command per data structure, whose different parts are later accessed by different processors. It is important to note that an allocation command on an operating system like Linux does not actually allocate virtual or even physical memory but address space. The allocation to virtual memory pages takes place when the memory is accessed the first time causing a page fault. Therefore, it is crucial that the first memory access on the data happens in the same pattern the processors will have during the rest of the calculations. The worst case occurs when the initialization is left sequentially and all the allocated memory is the root processor's local memory.

### *4.2. Computer architecture based issues*

Two high performance computers at the Steinbuch Centre for Computing at the University of Karlsruhe were considered as suitable prototypes for testing the realization of a hybrid parallelization.

The first one is an HP Integrity RX8620 16-way node which is partitioned into two 8-way (logical) nodes. Each Itanium-2 processor runs at a speed of 1.6 GHz and has 6 MB of level 3 cache. This (logical) node has 64 GB of main memory and one Quadrics QsNet II adapter and is part of a larger *HP XC6000* installation with another 100 2-way Itanium-2 nodes. Though eight processors share two memory banks, an interleaving mechanism provides uniform memory access where the aggregate memory bandwidth is 12.8 GB/s.

One two-way node (HP ProLiant DL145 G2) with two AMD Opteron sockets running at a clock speed of 2.6 GHz and equipped with 16 GB of main memory served as a second testing environment. Each socket hosts one dual-core processor with 1 MB of level 2 cache. The two cores of one processor share a memory bandwidth of 6.4 GB/s. The memory access is non-uniform, each socket has faster access to its own memory. The node provides a peak performance of 20.8 GFLOP/s and is also part of a large machine, the high performance computer *HP XC4000*, with 750 nodes, a peak performance of 15.6 TFLOP/s, 12 TB total main memory and an InfiniBand 4X DDR Interconnect.

### *4.3. Implementation details*

As part of the realization of a hybrid parallelization, OpenLB is parallelized for shared memory platform by using OpenMP. As mentioned previously in Section 3.4, pre-processor programming, an OpenMP manager and its global instance are provided to simplify the implementation. To enable a static assignment of a domain to a specific thread the instance is given for each thread and initialized with the number of dynamic threads set to zero.

If the collision and streaming step are executed separately in two loops over all `Cells` the data access is exclusive and the order of access arbitrary. It is stressed that this also holds for the streaming step due to a swapping technique [19–21] which is implemented in OpenLB. In a first attempt the collision (c) and streaming step (s) are parallelized straightforwardly using OpenMP directives for loops. Using the Intel compiler, it turns out that at both testing platforms (see Section 4.2) for the collision step a dynamic scheduling with block size one is the best choice, while for the streaming step the default scheduling is to be preferred. The OpenLB source code of the parallelized collision step is shown exemplarily for the presented straightforward approach in Fig. 2.

It is also possible to process the collision and streaming in one single loop (bulk c/s), even if the data of the distribution functions $f_i$ are stored only once in one array [19–21]. Unlike the previously described procedure the access order is not arbitrary. The execution of a `Cell` now depends on the data of itself and some particular `Cells` in the neighbourhood.

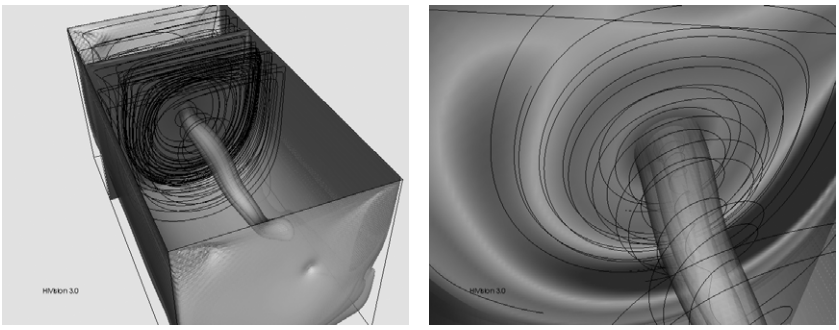

**Fig. 3.** Lid driven cavity (1:1:2): velocity distribution at $t = 12$, $Re = 1000$.

However there is no data dependency to `Cells` that are located somewhere in positive $x$-direction [21]. Using that specificity the domain of a `BlockLattice` is split along that direction into layers of as equal size as possible and each layer is assigned to a thread. To handle the correlation of the data the parallel bulk c/s is done in three steps. Firstly the lower boundary `Cells` of each layer perform a collision step. Secondly the layers without the lower boundary as new formed blocks execute a bulk c/s. Finally after synchronizing all threads the lower boundary `Cells` process a streaming step.

In OpenLB a class provides statistic data of a dedicated `BlockLattice` e.g. the average density, average energy and maximum velocity. A Method of this class is called while a `Cell` performs a collision, whereby it collects the statistic data and saves them in variables provided by an instance of the class. Since an exclusive access of one thread to an instance of the statistic class blockades the other threads, an approach using the corresponding OpenMP directive is inefficient. Additional provided instances of this class for each thread followed by a reduction solve that shortcoming satisfyingly. Thereby one has to take care to avoid that any two instances can be loaded into the same cache-line for this would also prevent access to the data for one thread as long as the other has loaded the line into his cache. This problem is solved by encapsulating each instance through allocating and assigning memory straight before and after an instance is constructed.

In order to improve the efficiency on multi-processor platforms with non-uniform memory access it is essential to ensure a static memory access. As pointed out in Section 4.1 data should be split into parts whereas each part is assigned to a specific thread and accessed preferably by this thread. In a second approach, the domain of a `BlockLattice` is split into blocks of as equal size as possible exactly as it is done for the bulk c/s described before. The OpenMP directive for loops with a static scheduling with the associated block size is used for the initialization of the data, the collision (c*) and streaming step (s*). The approach where the collision and streaming is done in one single loop remains unchanged and is now referred to as bulk c/s*. Threads are bind to specified cores by calling the Linux system function `sched_setaffinity()` to avoid the operating system moving the threads.

## 5. Numerical experiments and performance results

In order to validate the proposed parallel approach we consider the classical 3D benchmark problem lid driven cavity (LDC). To compare the numerical results with data gained experimentally the simulation setup is chosen exactly like the one of Guermond et al. presented in [22]. A Newtonian fluid in a rectangular cavity of ratio 1:1:2 is brought into motion by a constantly moving lid. The problem is characterized by the length and speed of the lid, each of them is assumed to be unity. In this system of units, the considered duration of the simulation is $\Delta t = 12$. The fluid is assumed to be incompressible with a Reynolds number of $Re = 1000$. In this configuration the fluid starts developing several vortices with one dominant right in the center of the cuboid. The pictures in Fig. 3 illustrate the described setup.

A D3Q19 lattice of size $201 \times 201 \times 401$ is used to solve the incompressible time-dependent 3D Navier–Stokes equations for this problem. This amounts to a total of about 308 million variables and requires, with the usage of IEEE double precision floating point arithmetic, a storage space of 2.46 GB in main memory. The Mach number is set to $Ma = 0.02/\sqrt{3}$, which is low enough to prevent compressibility effects to interfere with the accuracy of the result. With this setup of the numerical parameters, the characteristic length is 200 and the characteristic speed is 0.02. This yields 120 000 iteration steps for the system to reach the desired time of $\Delta t = 12$.

The simulation uses the single-relaxation-time BGK model, and the boundary condition is based on the technique suggested by Skordos in [23]. With this type of boundary conditions, velocity gradients on boundary nodes are evaluated using a second-order accurate finite difference scheme. This information is then used to compute the off-equilibrium terms of the particle distribution functions. It is also worth to note that on corner and edge nodes, the value of the pressure is extrapolated from bulk nodes, as the data locally available on the nodes is insufficient to evaluate the exact value of the pressure.

Due to the fact that the considered configuration is a closed vessel, the pressure distribution is defined up to a constant. Therefore adding a constant offset to the pressure does not modify the results of the incompressible Navier–Stokes equation. We observe however that the LBM experiences numerical instabilities when the pressure, and thus the particle density, is too high. For this reason, the average density in the system is computed at every iteration step of the simulation. At the next

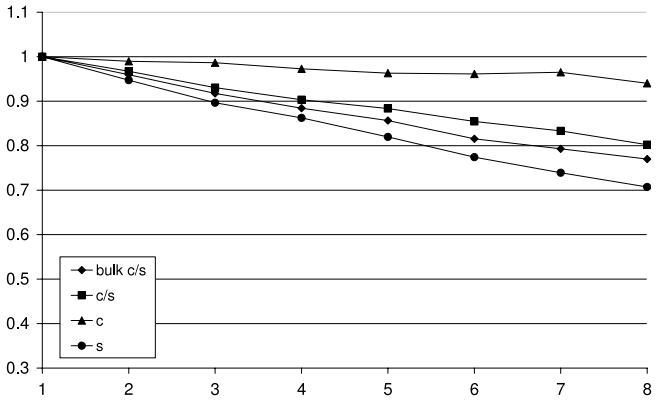


**Fig. 4.** Efficiency as a function of the number of threads on *HP XC6000*. The abbreviations in the legend are defined in Section 4.3.

**Table 1**
Computing times for 100 times steps on a 201 × 201 × 401 grid on *HP XC6000*. The abbreviations in the legend are defined in Section 4.3.

| Mode | Threads | Bulk c/s | c/s | c | s | Compiler flags |
|---|---|---|---|---|---|---|
| Sequential | 1 | 2405 s | 3006 s | 1359 s | 1628 s | -O3 |
| OpenMP | 1 | 2642 s | 3132 s | 1459 s | 1635 s | -O3-openmp |
| | 2 | 1377 s | 1619 s | 737 s | 863 s | |
| | 3 | 960 s | 1122 s | 493 s | 608 s | |
| | 4 | 747 s | 867 s | 375 s | 474 s | |
| | 5 | 617 s | 709 s | 303 s | 399 s | |
| | 6 | 540 s | 611 s | 253 s | 352 s | |
| | 7 | 476 s | 537 s | 216 s | 316 s | |
| | 8 | 429 s | 488 s | 194 s | 289 s | |

iteration, a constant offset is subtracted from the density on every grid node, in order to keep the average density close to a value of 1.

This simulation is intended to benchmark multithreaded performance of the OpenLB code. It is executed on the multiprocessor platform with 4 cores (*HP XC4000*), on which it takes 4 days and 12 h to complete. Comparing the positions of the main vortex and two smaller secondary eddies gained by the simulation with the experimentally gained positions at three chosen planes within the cavity, the deviation is found to be within 4.2%. Detailed results regarding accuracy can be found in [24].

To get reliable performance results, 100 time steps are considered for the testing and benchmarking. Since the computation and copying operations are identical for every single time step an examination of this number of time steps is sufficient to get scalable results. All source code is compiled with the Intel compiler using optimization level 3. For sequential and parallel modes the compiler uses two different optimization settings for performance tuning. It is important to note that this optimization setting involves specific heuristics especially to control the use of memory bandwidth among processors. It turns out that this setting leads to an increase of the overall computing times on a single thread. Each configuration is executed at least three times to resolve random, possibly hardware caused artefacts polluting the presented results, whereby the best one of them is presented in the following.

The performances related to the first approach described in Section 4.3 are presented in Table 1 and Fig. 4 for the computations on the platform *HP XC6000* (resp. in Table 2 and Fig. 5 for the computations on the platform *HP XC4000*). The results obtained on the platform with uniform memory access (*HP XC6000*) show a smooth and almost linear decrease of the efficiency as a function of threads, leading to an efficiency of approximately $E_{ff} \approx 0.80$ for eight threads. While the performance results of the three tests on this platform are found with small variations, the results of the first approach obtained on the platform without uniform memory access (*HP XC4000*) show variations of up to 20%. Moreover the reached efficiency is much smaller. These results are in accordance with numerical tests obtained on other hardware platforms by means of OpenMP for LBM [15,25].

A comparison of the efficiency on the platform *HP XC4000*, assuming a local memory allocation and a local first touch of data, is depicted in Fig. 5. The efficiency can be improved significantly and the variation of the test results is found to be less than five per cent. The best result for that approach is achieved for the bulk c/s* using two threads, reaching an efficiency of $E_{ff} = 0.95$, while the first approach only leads to $E_{ff} = 0.82$.

Apart from a few exceptions it is observed on both platforms that performing the collision and streaming step in one single loop over all `Cells` (bulk c/s) is more efficient, considering the absolute execution times, than in two separated loops (c/s). This is pointed out as well for other platforms in [19,20]. It is emphasized that this holds for both the sequential and parallel implementation for the presented benchmark problem.

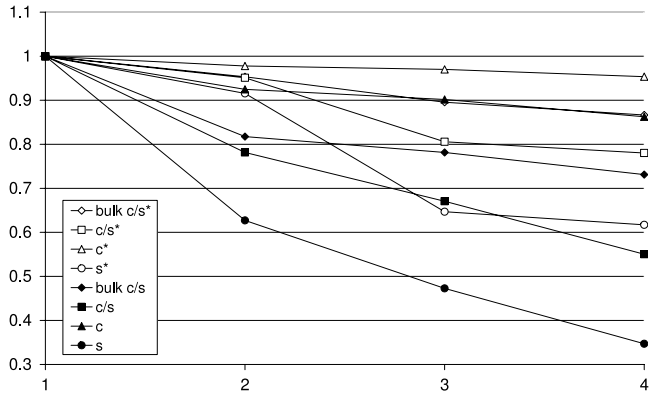


**Fig. 5.** Comparison of the efficiency as a function of the number of threads on the platform *HP XC4000*. The two approaches and related abbreviations in the legend are declared in Section 4.3. The abbreviations marked with an asterisk belong to the approach at which memory allocation and first touch of data is done locally and the threads are bound to specified cores.

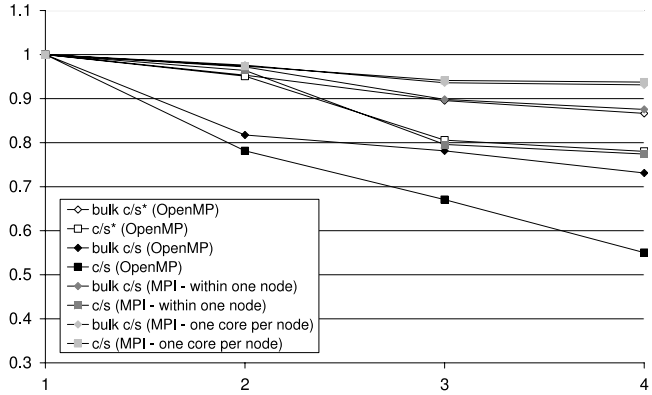


**Fig. 6.** Efficiency as a function of the number of threads on *HP XC4000* in comparison with the results of a MPI based parallelization. The abbreviations in the legend are defined in Section 4.3.

**Table 2**
Computing times for 100 times steps on a $201 \times 201 \times 401$ grid on *HP XC4000*. The abbreviations in the legend are defined in Section 4.3.

| Mode | Threads | Bulk c/s | c/s | c | s | Compiler flags |
|---|---|---|---|---|---|---|
| Sequential | 1 | 902 s | 947 s | 589 s | 330 s | -O3 |
| OpenMP | 1 | 1123 s | 1008 s | 614 s | 390 s | -O3-openmp |
| OpenMP | 2 | 687 s | 645 s | 332 s | 311 s | |
| | 3 | 479 s | 501 s | 227 s | 275 s | |
| | 4 | 384 s | 458 s | 178 s | 281 s | |
| OpenMP (local memory) | 2 | 589 s | 530 s | 314 s | 213 s | |
| | 3 | 418 s | 417 s | 211 s | 201 s | |
| | 4 | 324 s | 323 s | 161 s | 158 s | |
| MPI | 1 | 935 s | 960 s | 597 s | 347 s | -O3 |
| MPI (within one node) | 2 | 481 s | 498 s | 293 s | 195 s | |
| | 3 | 347 s | 402 s | 200 s | 204 s | |
| | 4 | 267 s | 310 s | 150 s | 156 s | |
| MPI (one core per node) | 2 | 479 s | 493 s | 293 s | 192 s | |
| | 3 | 333 s | 340 s | 195 s | 141 s | |
| | 4 | 251 s | 256 s | 145 s | 105 s | |

In Fig. 6 the efficiency of a purely MPI based approach is presented and compared to the corresponding results of the OpenMP versions. Remark that the considered implementation tested on one node uses a direct access to the local shared memory. It is important to notice that for the case of three and four processes the version executed on three respectively four different nodes i.e. involving extranodal communication is more efficient than the version involving intranodal communication. In the case of two processes there is no significant difference to observe. Employing only one core per socket, then each core can almost saturate the memory-to-socket bandwidth in contrast to the case of three and four cores per node and thus the limitation of the bandwidth becomes visible. Comparing the efficiency based on the overall

execution time using the same code executed on one core, the OpenMP version with the local memory assignment and the MPI version are found to be of high conformance. Though, considering the absolute execution times as they are presented in Table 2, it is observed that the MPI version is more efficient than both OpenMP based approaches. Moreover the MPI results clearly show the intricate dependency between the efficiency and job partitioning at the core and nodal level. The motivation of the proposed hybrid parallelization concept relies on the idea that this kind of local partition at the nodal level should automatically be treated by the considered parallelization paradigm. The authors are convinced that OpenMP offers such an interface which obviously still needs to be optimized.

## 6. Conclusions

This paper introduces a framework for a hybrid parallelization of lattice Boltzmann code. The presented concept is realized within the C++ project OpenLB. The implementation is based on a partition of a domain into blocks which are distributed across the nodes of a cluster. The communication between the blocks occurs by means of MPI whereas the parallelization of a block is handled considering OpenMP on SMP nodes.

The presented benchmarks show that the distributed memory paradigm based on MPI reaches higher efficiency than the shared memory approach of OpenMP even on SMP nodes. These somewhat disappointing results are partly due to heuristics used by the compiler when using OpenMP leading to a loss in efficiency even for the case of sequential execution. To achieve better performance results the memory hierarchy and moreover on platforms without uniform memory access a binding of threads to specific processors have to be taken into account. Since OpenMP does not provide any support for memory hierarchies and to control affinity, the programmer is left to use utilities provided by the operating environment.

Multi-core platforms are however a reality which cannot be circumvented, and whose importance is expected to increase strongly in the near future. It can also be expected that the needs of high performance computing are better taken into account in future hardware platforms, as the communication between cores is improved. Hybrid implementations as the one presented in this paper are therefore of crucial importance. We believe that the approach described here takes into account the fundamental philosophy of lattice Boltzmann methods. It combines efficiency with ease of use and could serve as a programming paradigm for lattice Boltzmann implementations on current and on future computation platforms.

## Acknowledgements

The authors greatly thank Christian Terboven from the Center for Computing and Communication of RWTH Aachen University for his kind support and many important suggestions in the context of OpenMP programming. With respect to code optimization which is closely related to the underlying hardware of both considered platforms, the authors greatly acknowledge Werner Augustin from the Steinbuch Centre for Computing (SCC) at the University of Karlsruhe (TH) for his support and strong commitment.

## References

[1] M.C. Sukop, D.T. Thorne, Lattice Boltzmann Modeling, Springer, 2006.
[2] B. Chopard, M. Droz, Cellular Automata Modeling of Physical Systems, Cambridge University Press, 1998.
[3] D. Hänel, Molekulare Gasdynamik, Springer, 2004.
[4] S. Chen, G. Doolen, Lattice Boltzmann method for fluid flows, Annual Review of Fluid Mechanics 30 (1998) 329–364.
[5] V. Heuveline, M.J. Krause, J. Latt, O. Malaspinas, Open source lattice Boltzmann code (OpenLB). http://www.openlb.org.
[6] S. Succi, G. Amati, R. Benzi, Challenges in lattice Boltzmann computing, Journal of Statistical Physics 81 (1995) 5.
[7] M. Krafczyk, E. Rank, A parallelized lattice-gas solver for transient Navier–Stokes-flow: Implementation and simulation results, Journal for Numerical Methods in Engineering 38 (1995) 1243–1258.
[8] J. Latt, B. Chopard, Vladymir — a C++ matrix library for data-parallel applications, Future Generation Computer Systems 20 (2004) 1023–1039.
[9] S. Geller, M. Krafczyk, J. Tölke, S. Turek, J. Hron, Benchmark computations based on lattice-Boltzmann, finite element and finite volume methods for laminar flows, Computers and Fluids 35 (2004) 888–897.
[10] A. Dupuis, B. Chopard, An object oriented approach to lattice gas modeling, Future Generation Computer Systems 16 (2000) 523–532.
[11] G. Wellein, T. Zeiser, S. Donath, G. Hager, On the single processor performance of simple lattice Boltzmann kernels, Computers and Fluids 35 (8–9) (2006) 910–919.
[12] J. Ni, C.-L. Lin, Y. Zhang, T. He, S. Wang, B. Knosp, Parallelism of lattice Boltzmann method (LBM) for lid-driven cavity flows, in: Proceedings of High Performance Computing and Applications, HPCA2004, August 8-10, 2004, Shanghai, PR China.
[13] F. Massaioli, G. Amati, Achieving high performance in a LBM code using OpenMP, in: EWOMP 2002, 2002.
[14] T. Pohl, F. Deserno, N. Thurey, U. Rude, P. Lammers, G. Wellein, T. Zeiser, Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures, in: Supercomputing 2004 (Proceedings of the ACM/IEEE SC2004 Conference), 2004, p. 21.
[15] T. Zeiser, J. Götz, 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 more complex (physical) models and changing trends in high performance computing!? in: E. Krause, Y.I. Shokin, M. Resch, N. Shokina (Eds.), Computational Science and High Performance Computing III (The 3rd Russian-German Advanced Research Workshop, Novosibirsk, Russia, 23-27 July 2007), in: Notes on Numerical Fluid Mechanics and Multidisciplinary Design, vol. 101, Springer, ISBN: 978-3-540-69008-5, 2008, pp. 165–183. doi:10.1007/978-3-540-69010-8_13.
[16] V. Heuveline, J.-P. Weiß, A parallel implementation of a lattice Boltzmann method on the ClearSpeed Advance Accelerator Board, IWRMM-Preprints, Nr. 07/03, 2007.
[17] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, Parallel Programming in OpenMP, Morgan Kaufmann, 2001.
[18] J. Bull, Measuring synchronisation and scheduling overheads in OpenMP, in: European Workshop on OpenMP, EWOMP1999, Lund, Sweden, 1999.
[19] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, J. Westerholm, An efficient swap algorithm for the lattice Boltzmann method, Computer Physics Communications 176 (2007) 200–210.

[20] K. Mattila, J. Hyväluoma, J. Timonen, T. Rossi, Comparison of implementations of the lattice-Boltzmann method, Computers & Mathematics with Applications (2007) doi:10.1016/j.camwa.2007.08.001.
[21] J. Latt, How to implement your DdQq dynamics with only q variables per node (instead of 2q), Technical Report, Tufts University Medford, USA, 2007.
[22] J.-L. Guermond, C. Migeon, G. Pineau, L. Quartapelle, Start-up flows in a three-dimensional rectangular driven cavity of aspect ration 1:1:2 at $Re = 1000$, Journal of Fluid Mechanics 450 (2002) 169–199.
[23] P. Skordos, Initial and boundary conditions for the lattice Boltzmann method, Physical Review E 48 (6) (1993) 4823–4842.
[24] H. Bockelmann, V. Heuveline, M.J. Krause, Fluid flow simulations using lattice Boltzmann and finite element methods: A comparison (2007) (in preparation).
[25] T. Zeiser, T.G. Wellein, A. Nitsure, K. Iglberger, U. Rüde, G. Hager, Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method, Progress in Computer Fluid Dynamics (2006).