Available online at www.sciencedirect.com

ScienceDirect

Computer Physics Communications 176 (2007) 200–210

Computer Physics Communications

www.elsevier.com/locate/cpc

An efficient swap algorithm for the lattice Boltzmann method

Keijo Mattila [a,b,*], Jari Hyväluoma [b], Tuomo Rossi [a], Mats Aspnäs [c], Jan Westerholm [c]

[a] *Department of Mathematical Information Technology, University of Jyväskylä, P.O. Box 35, FI-40014 Jyväskylä, Finland*
[b] *Department of Physics, University of Jyväskylä, P.O. Box 35, FI-40014 Jyväskylä, Finland*
[c] *Department of Computer Science, Åbo Akademi University, Lemminkäinengatan 14 A, FI-20520 Åbo, Finland*

Received 14 February 2006; received in revised form 19 September 2006; accepted 28 September 2006

Available online 27 October 2006

Abstract

During the last decade, the lattice-Boltzmann method (LBM) as a valuable tool in computational fluid dynamics has been increasingly acknowledged. The widespread application of LBM is partly due to the simplicity of its coding. The most well-known algorithms for the implementation of the standard lattice-Boltzmann equation (LBE) are the two-lattice and two-step algorithms. However, implementations of the two-lattice or the two-step algorithm suffer from high memory consumption or poor computational performance, respectively. Ultimately, the computing resources available decide which of the two disadvantages is more critical. Here we introduce a new algorithm, called the swap algorithm, for the implementation of LBE. Simulation results demonstrate that implementations based on the swap algorithm can achieve high computational performance and have very low memory consumption. Furthermore, we show how the performance of its implementations can be further improved by code optimization.
© 2006 Elsevier B.V. All rights reserved.

PACS: 02.60.Cb; 02.70.Ns; 47.11.+j

Keywords: Lattice Boltzmann; Algorithm; Implementation; Code optimization

1. Introduction

Historically, the lattice-Boltzmann method has emerged from the lattice-gas automata in the late 1980s [1–3]. Since then LBM has been applied to a multitude of fluid flow problems. In particular, LBM is used in the fields of multiphase flows, suspension flows, and fluid flows in porous media [3]. The main assets of LBM include simplicity of coding, straightforward incorporation of microscopic interactions, and suitability for parallel computing. More debatable is the computational efficiency of LBM in comparison to other methods, such as finite-element and finite-volume methods. This issue remains to be resolved. According to recent studies, LBM is competitive for incompressible transient problems, but asymptotically slower for steady-state Stokes flows [4].

A low convergence rate to steady-state is a widely accepted defect of the standard LBM, and there have been efforts to accelerate this saturation process (see Ref. [5] and references therein). Some general methods to achieve high computational efficiency, or performance, with LBM are grid refinement [6], improved algorithms for explicit time-marching implementations [7,8], and code optimization [8–10]. The aforementioned methods are not restricted to steady-state flows, and they can also be applied to transient problems.

Here we consider implementations of LBM that achieve high computational performance with low memory consumption. Specifically, we consider explicit time-marching implementations of LBM on a uniform lattice. Our focus is on applications that require very large simulation domains—as measured by the number of lattice nodes. We assume that a significant volume fraction is occupied by solid structures. A typical example of such an application is flow in porous media such as paper and sandstone. Furthermore, we only consider here single-phase flows. Notice also that the geometry of the simulation domain is assumed immobile. This facilitates particular implementation techniques.

[*] Corresponding author.
E-mail address: kemattil@cc.jyu.fi (K. Mattila).

0010-4655/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2006.09.005

Argentini et al. presented an implementation for LBM which achieves a very low memory consumption [11]. In their implementation, only certain hydrodynamic moments are stored for each fluid lattice node. This is a viable approach as they choose to utilize a particular collision operator. Due to this collision operator, the post-collision distribution values can be written in terms of the stored moments of the pre-collision distribution values. In a very special case, the non-linear term in the equilibrium momentum flux tensor is neglected and the relaxation parameter $\lambda = -1$. Then it is necessary to store only density and components of the velocity vector for each fluid node. Here, instead of storing certain moments of the distribution values, we follow the traditional approach, and store the distribution values for fluid nodes.

In Ref. [12], the single relaxation time scheme BGK was utilized as a collision operator. They also assumed instant relaxation to the local equilibrium, i.e. $\lambda = 1$. This can be exploited in the implementation, resulting in very low memory consumption: again it is necessary to store only the densities and the components of the velocity vectors for the fluid nodes. Furthermore, Martys et al. utilized so-called semi-direct memory addressing where memory is not allocated for the distribution values of the solid nodes. Semi-direct addressing is utilized also in this work.

Indirect memory addressing schemes for implementations of LBM were investigated in Refs. [13–15]. They all concluded that there is a threshold for fluid volume fraction below which indirect addressing consumes less memory than direct addressing. It was shown that with very low fluid volume fractions, the memory savings can be considerable. Also, according to Schulz et al., implementations of the two-lattice algorithm have twice the computational performance than implementations of the two-step algorithm. As far as we know, relative memory consumptions of semi-direct and indirect addressing schemes remains to be investigated. Even so, relative memory consumptions of addressing schemes are not investigated here.

Massaioli et al. proposed an algorithm for implementation of LBM based on Lagrangian approach [7]. They compared their new algorithm against the two-step algorithm, and slight performance improvement was observed. In order to overcome the high memory consumption of the two-lattice algorithm, Pohl et al. introduced the compressed grid (shift) algorithm, and gave some benchmarking results including comparison with the two-lattice algorithm [8]. They found that, with the shift algorithm, it is possible to attain computational performance of the two-lattice algorithm, and at the same time almost halve the memory requirements. Pohl et al. also considered some code optimization techniques, such as blocking. Recently, data layouts for distribution values were considered in Ref. [10]. In particular, the effect of data layouts on computational performance was studied. Wellein et al. concluded that correct choice of the data layout can be fundamental for achieving high performance. In this work, collision optimized data layout is used.

The main motivation of this work is to introduce a new algorithm, called the swap algorithm, for the implementation of LBM. Computational performances are compared, and presented for the shift, swap, two-lattice, and two-step algorithms. Also, the relative memory consumptions of the above algorithms are observed in a simple channel geometry. Furthermore, we show how the performance of the implementations can be further improved by code optimization. Both algorithms and code optimization can be applied in conjunction with many other acceleration techniques. The code optimizations reported here are not designed for any particular processor or system architecture. Instead, the optimizations are generally applicable to modern processors, and aim at improving the cache utilization and providing more opportunities for instruction-level parallelism.

In Section 2, we give a short introduction to LBM. Implementation aspects are discussed in Section 3. The shift algorithm is described in Section 4. Also, the novel swap algorithm is introduced in the same section. Results of our numerical experiments as well as comparison of the algorithms are presented in Section 5. Some code optimization techniques are investigated in Section 6. Finally, conclusions are drawn in Section 7.

2. Lattice-Boltzmann method

LBM is a specific finite-difference discretization of the continuous Boltzmann equation [16–18]. In LBM, both time and phase space are discrete. From its ancestors, the lattice-gas automata, an evocative description is inherited for LBM: starting from an initial configuration, a system of fictitious mesoscopic particles evolves on a lattice. During one time step, the particles move from their lattice nodes to neighboring nodes according to their velocities. When several particles arrive at the same lattice node, they collide with each other and change their velocities. The average motion of the particles describes the macroscopic behavior of the system.

The state of the system is defined by the single-particle distribution functions $f_i(\vec{r}, t)$ indicating the probability of finding a particle at site $\vec{r}$ at time t with velocity $\vec{c}_i$. Here and hereafter $\vec{r}$, t, and $\vec{c}_i$ are expressed in lattice units. The dynamics of the system is governed by the lattice-Boltzmann equation (LBE)

$$f_i(\vec{r} + \vec{c}_i, t + 1) = f_i(\vec{r}, t) + \Omega_i\big(f(\vec{r}, t)\big), \quad i = 0, \dots, b - 1, \tag{1}$$

where Ω_i is the collision operator and b is the number of possible velocities for the fictitious particles. With $\Omega_i(f(\vec{r}, t))$ we emphasize the dependence of the collision operator on all b distribution functions associated with the site $\vec{r}$ at time t.

There are many choices for the collision operator Ω_i in Eq. (1). The most simple one is the single-relaxation-time approximation. In kinetic theory, this is known as the BGK approximation after Bhatnagar et al. [19]. Applying the BGK approximation in Eq. (1), one obtains the lattice-BGK (LBGK) model:

$$f_i(\vec{r} + \vec{c}_i, t + 1) = f_i(\vec{r}, t) + \frac{1}{\tau}\big(f_i^{(\mathrm{eq})}(\vec{r}, t) - f_i(\vec{r}, t)\big). \tag{2}$$

The parameter τ is the relaxation time characterizing the collision process by which the distribution functions relax towards their equilibrium values $f_i^{(\mathrm{eq})}(\vec{r}, t)$. The basic hydrodynamic quantities of the fluid, the mass density ρ and the momentum density $\rho\vec{u}$, are calculated from the zeroth and first moments of the distribution functions:

$$\rho(\vec{r},t)=\sum_i f_i(\vec{r},t),$$
$$\rho(\vec{r},t)\vec{u}(\vec{r},t)=\sum_i \vec{c}_i f_i(\vec{r},t).$$

A family of LBGK models for computational fluid dynamics is presented in Ref. [20]. These models have b discrete velocities on a simple cubic lattice of dimension d (DdQb models). The equilibrium distribution function for these models is

$$f_i^{(\mathrm{eq})}(\vec{r},t)=w_p\rho\left(1+\frac{(\vec{c}_i\cdot\vec{u})}{c_s^2}+\frac{(\vec{c}_i\cdot\vec{u})^2}{2c_s^4}-\frac{u^2}{2c_s^2}\right). \tag{3}$$

In Eq. (3), c_s is the speed of sound, the index p refers to the square modulus of the velocity $\vec{c}_i$ and w_p is the corresponding equilibrium distribution for $u=0$. For these models, the speed of sound and the kinematic viscosity are given by

$$c_s=\frac{1}{\sqrt{3}},\qquad \nu=\frac{2\tau-1}{6}.$$

The pressure is related to the fluid density by the equation of state: $p(\vec{r},t)=c_s^2\rho(\vec{r},t)$. For example, in the D3Q19 model which is used in the numerical work of this paper, $w_0=1/3$, $w_1=1/18$ and $w_2=1/36$; the velocity vectors $\vec{c}_i$ are $(0,0,0)$, $(\pm1,0,0)$, $(\pm1,\pm1,0)$ and permutations thereof. Through the Chapman–Enskog analysis, the incompressible Navier–Stokes equations can be recovered from LBE in the low Mach-number limit [21–23].

An attractive feature of LBM is the easy implementation of the fluid–solid no-slip boundary conditions. This facilitates flow simulations in complex geometries. The simplest method to realize the no-slip boundary condition is the heuristic bounce-back rule where the momenta of the fluid particles colliding with a wall are reversed [21,24]. Many other boundary conditions have also been proposed, such as the interpolation-based scheme introduced in Ref. [25].

3. Implementation of LBM

In this paper we consider explicit time-marching implementations of LBM. We limit our investigation to implementations based on semi-direct addressing scheme. In this scheme neighboring lattice nodes are implicitly known for every node through the structure of the lattice, e.g., from node indexing. More importantly, no memory is allocated for the distribution values of the lattice nodes inside solid structures. Beyond our investigation are implementations utilizing indirect addressing, where information about the neighboring lattice nodes must be explicitly provided [13–15]. For example, indirect addressing can be appropriate for simulations with local grid refinements, very low fluid-volume fractions, or elaborate domain decompositions.

Here we introduce some definitions needed in the following discussion. Albeit the concepts are made concrete with two-dimensional examples, they also apply in three dimensions. Let a two-dimensional square lattice $\mathcal{L}$ include the actual computational domain $\mathcal{D}$, and a ghost layer $\mathcal{G}=\mathcal{L}\setminus\mathcal{D}$ surrounding the computational domain:

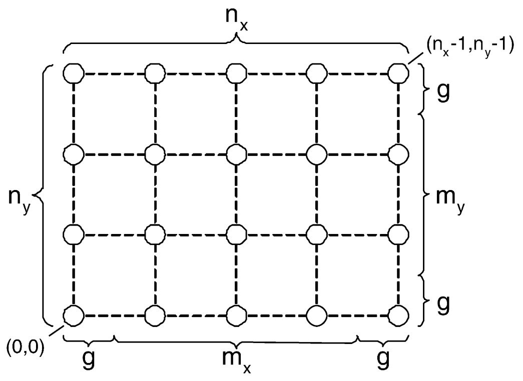

Fig. 1. A two-dimensional lattice including ghost layers: $n_x=5, n_y=4$ and $g=1$.

$$\mathcal{L}=[0,n_x-1]\times[0,n_y-1]\subset\mathbb{N}^2,$$
$$\mathcal{D}=[g,n_x-g-1]\times[g,n_y-g-1]\subset\mathcal{L},$$

where g is the width of the ghost layer expressed in lattice nodes. We assume that the width of the ghost layer is compatible with the velocity set, i.e. $(\vec{d}+\vec{c}_i)\in\mathcal{L}$ holds true for every $\vec{d}\in\mathcal{D}$. For the computational domain, the number of lattice nodes in x and y directions are denoted by m_x and m_y, respectively. The dimensions of the lattice $\mathcal{L}$ are then $n_x=2g+m_x$ and $n_y=2g+m_y$ (cf. Fig. 1). The ghost layer is not mandatory for LBM implementations but merely a practical instrument, which we include for convenience. For example, it can be used to enforce the boundary conditions for the computational domain and to simplify the implementation of the streaming step.

In order to have an efficient implementation on modern computers, a high utilization of the cache memory hierarchy must be ensured. This, in turn, requires that data layout, data order, and data update procedures are in harmony with each other. To lay the ground for efficient LBM implementations, we define the data order as follows. We choose to store all distribution values of the lattice in a vector $\vec{V}$, i.e. we vectorize the data. On that account, we enumerate the lattice nodes. For a two-dimensional square lattice $\mathcal{L}$, we define the enumeration function

$$n(\vec{l})=l_y\cdot n_x+l_x,\quad \vec{l}\in\mathcal{L}. \tag{4}$$

The function $n:\mathcal{L}\to\mathbb{N}$ assigns a unique positive integer to each lattice node and thus defines the data order in the vector $\vec{V}$.

The definition of the data structure $\vec{V}$ is still incomplete; we should also define how the distribution values f_i of each node are stored in $\vec{V}$. We choose to assemble all the distribution values of a lattice node and store them consecutively in the vector $\vec{V}$. This is known as the collision optimized data layout [10]. There are many alternatives for the data layout: for example, the propagation optimized data layout. In this work, we do not consider the effect of data layouts on the computational performance. For a lattice node $\vec{l}$, the order of the distribution values $f_i(\vec{l},t)$ in $\vec{V}$ is congruent with that of velocity vectors.

The order of the velocity vectors is fixed implicitly by demanding that a set of constraints $n(\vec{d}+\vec{c}_i)<n(\vec{d}+\vec{c}_{i+1})$,

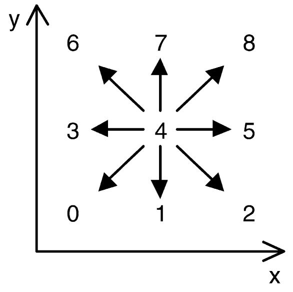

Fig. 2. Enumeration of the velocity vectors for the D2Q9 model.

$\vec{d} \in \mathcal{D}, i = 0, \ldots, b-2$, is fulfilled. This choice is motivated by the fact that if lattice nodes are iterated in the order defined by the enumeration function, e.g., by Eq. (4), there is a rigorous order in which the distribution values of a particular node are accessed—namely the one defined by the above constraints. For the D2Q9 model [20], and with the enumeration function (4), Fig. 2 describes the enumeration of the velocity vectors as defined by the above constraints.

As a case study we consider single-phase flow in a system that may contain static solid structures. In other words, every lattice node is either a fluid node or a solid node. A typical example of this kind of application is the flow in a porous medium, a problem to which LBM has been successfully applied (see e.g. [26]). We introduce a vector $\vec{P}$ in which the phase information is stored for each node in $\mathcal{L}$. If the geometry of the computational domain is static, like in our case study, it is rather easy to reduce the memory consumption: no memory is allocated for the distribution functions of the solid nodes [12]. To this end, we define an exclusive enumeration for the fluid nodes. Let sets $\mathcal{F}$, $\mathcal{F}_{\mathcal{D}}$, and $\mathcal{F}_{\mathcal{G}}$ include the fluid nodes in $\mathcal{L}$, $\mathcal{D}$, and $\mathcal{G}$, respectively. A mapping $n_f : \mathcal{F} \to \mathbb{N}$ associates a unique positive integer from the range $[0, N_f - 1]$ to each fluid node, where N_f is the number of fluid nodes in the lattice. The mapping n_f preserves the original order of the fluid nodes, i.e.

$$\big(n(\vec{l}_1) < n(\vec{l}_2)\big) \quad \Rightarrow \quad \big(n_f(\vec{l}_1) < n_f(\vec{l}_2)\big), \quad \vec{l}_1, \vec{l}_2 \in \mathcal{F}.$$

In our case study, we use semi-direct addressing. In such a scheme, distribution values of adjacent fluid nodes are accessed not only through the lattice structure, but also by using additional addressing information like indices or pointers [12]. The mapping n_f provides the additional addressing information and the vector $\vec{P}$ is employed as a storage for the order of the fluid nodes. If the lattice node $\vec{l}$ is a solid node, then the corresponding element $n(\vec{l})$ in the vector $\vec{P}$ has a negative value, say -1. Otherwise, the element contains the number $n_f(\vec{l})$.

4. Algorithms

The time evolution in explicit time-marching implementations of LBM results from the alternation of two distinct steps, streaming and collision. Expressions for these steps can be obtained by splitting Eq. (1) formally in time. The resulting streaming step is non-local, as particles or values of the distribution functions propagate between adjacent lattice nodes according to

$$f_i(\vec{r} + \vec{c}_i, t^\star) = f_i(\vec{r}, t). \tag{5}$$

The completely local collision step is

$$f_i(\vec{r} + \vec{c}_i, t+1) = f_i(\vec{r} + \vec{c}_i, t^\star) + \Omega_i\big(f(\vec{r} + \vec{c}_i, t^\star)\big). \tag{6}$$

The difference between equations above and Eq. (1) is that in the latter, at each time step, collision precedes streaming while Eqs. (5) and (6) advocate the opposite order. Collision preceding streaming and vice versa correspond to "push" and "pull" schemes, respectively [10]. The "pull" scheme is adopted for the algorithms presented in this paper.

The streaming step in Eq. (5) gives rise to a coupling between the distribution values of adjacent lattice nodes. There are two elementary algorithms to deal with this kind of data dependence. As the first elementary algorithm we consider the two-step algorithm where the distribution values are updated in two separate steps [3,9,27]. To begin with, the streaming step is performed for the whole lattice. In order to ensure that the distribution values are updated from the appropriate values in the previous time step, the distribution values are propagated into adjacent lattice nodes in a carefully selected order. The streaming step can be implemented conveniently with the aid of the ghost layer surrounding the computational domain. After the streaming step has been performed for every lattice node, the algorithm advances to the collision step. These two separate steps at the implementation level implies that the lattice must be traversed at least twice at each time step. This creates an overhead in the simulation time mainly due to an increase in cache misses and the instruction count.

The second elementary algorithm we consider is the two-lattice algorithm [8,27]. As the name implies, in implementations based on the two-lattice algorithm the memory allocated for the distribution values is duplicated. At even time steps, the distribution values are read from the first memory addresses and stored in the second addresses. At odd time steps, the procedure is reversed. In the two-lattice algorithm, the streaming step and the collision step can be fused at the implementation level. Typically fused implementations outperform the non-fused ones [27]. Even though the implementation of the two-lattice algorithm is straightforward, it consumes nearly twice the amount of memory compared to implementations of the two-step algorithm. An extravagant usage of memory is unacceptable in the case of large lattices, and thus more economical implementations are needed. Where memory consumption is the main concern, and some special circumstances prevail, Refs. [11,12] present methods to reduce the memory requirements considerably. In the method proposed in Ref. [11], a particular collision operator must be utilized whereas in Ref. [12], an instant relaxation to the local equilibrium is assumed.

Next we consider two algorithms, both of which can be used to produce an efficient implementation of LBM with low memory consumption. Both algorithms, the shift and the swap algorithm, are based on the Eulerian approach and in both the streaming step is fused with the collision step at the implementation level. In the Eulerian approach, the streaming step is executed explicitly unlike in the Lagrangian approach where

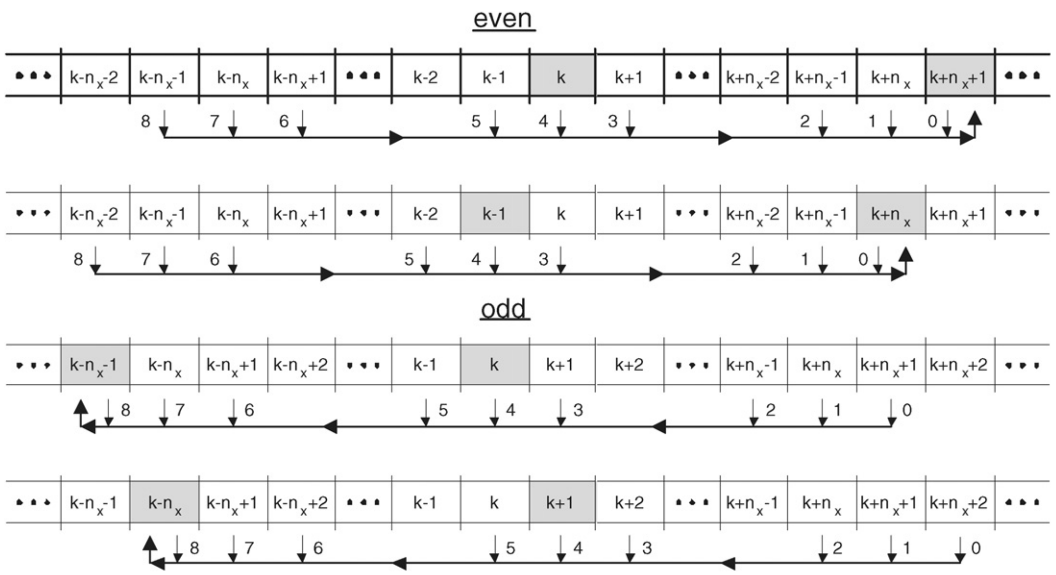

Fig. 3. In the shift algorithm, distribution values are shifted in $\vec{V}$ from left to right at even time steps, and from right to left at odd time steps.

the streaming step is an implicit operation. An algorithm based on the Lagrangian approach, with fused implementation, is presented in Ref. [7].

4.1. Shift algorithm

An innovative and efficient algorithm for LBM was introduced by Pohl et al. [8]. Their algorithm is a descendant of the two-lattice algorithm, and they apply an ingenious data compression technique to reduce the memory consumption of implementations based on the two-lattice algorithm. We refer to their algorithm as the shift algorithm for reasons which will become apparent. Like the two-lattice algorithm, the shift algorithm also takes advantage of two memory addresses per distribution value. As we choose to store all distribution values in the vector $\vec{V}$, the memory addresses are associated with the elements of $\vec{V}$. In general, this needs not to be the case.

Two sets of memory addresses can be constructed such that every distribution value has one memory address from both sets. In the two-lattice algorithm the two sets of memory addresses are distinct whereas in the shift algorithm the sets intersect. Evidently, the memory savings introduced by the shift algorithm are proportional to the overlap of the sets. This overlap is determined by the data dependences involved in the implementation. Let S represent the spatial extent of the data dependences in lattice units for an arbitrary geometry. For the D2Q9 model in conjunction with the bounce-back boundary conditions or with boundary conditions based on second-order interpolations like the one proposed in Ref. [25], appropriate values are $S = 1$ and $S = 2$, respectively.

In the shift algorithm, the data dependences can be acknowledged by adopting a strict iteration order for the lattice nodes, and by shifting the distribution values between successive time steps. To enable the shifting, the vector $\vec{V}$ is extended, having a total number of $N_V = (N_f + N_e) \cdot b$ elements. The offset $N_e = n(\vec{s})$ is calculated by evaluating the enumeration function with the vector $\vec{s} = (S, S)$. For example, with enumeration function (4) and $S = 1$, the offset is $N_e = n_x + 1$ (see Fig. 3).

At even time steps the shift algorithm proceeds by iterating the lattice nodes in the domain $\mathcal{D}$ starting from the node having the largest enumeration number down to the node with the smallest number. For every fluid node $\vec{d}$, all the distribution values participating in the streaming step are read from the vector $\vec{V}$ by using the enumeration number $n_f(\vec{l})$, where $\vec{l}$ is the fluid node from which the particular distribution value originates. The updated values are stored with the enumeration number $n_f(\vec{d}) + N_e$. At odd time steps, iteration is started from the node with smallest enumeration number; contrary to the even time steps, enumeration numbers $n_f(\vec{l}) + N_e$ are used for reading and $n_f(\vec{d})$ for storing. In brief, at even time steps distribution values are shifted from left to right. For odd time steps, values are shifted in the opposite direction, as depicted in Fig. 3. Implementations of the two-lattice algorithm consume memory almost twice as much as implementations based on the shift algorithm.

Let $\{\vec{d}_0, \vec{d}_1, \ldots, \vec{d}_{N-1}\}$ be an ordered set of all lattice nodes in the computational domain $\mathcal{D}$. The order is defined by the enumeration number: $n(\vec{d}_{k-1}) < n(\vec{d}_k)$, $k = 1, \ldots, N-1$. For the shift algorithm, the main body of the implementation can be described as follows:

```
$read_{\text{offset}} := 0$; $write_{\text{offset}} := N_e$
$k_{\text{start}} := N - 1$; $k_{\text{end}} := 0$
for $t := 0$ to $(iters - 1)$ do
  enforce_boundary_conditions($read_{\text{offset}}$)
  for $k := k_{\text{start}}$ to $k_{\text{end}}$ do
    $e :=$ element $n(\vec{d}_k)$ of $\vec{P}$
    if $e \geqslant 0$ then
      stream_and_collide($k, e, read_{\text{offset}}, write_{\text{offset}}$)
  end
  exchange($read_{\text{offset}}, write_{\text{offset}}$); exchange($k_{\text{start}}, k_{\text{end}}$)
end
```

4.2. Swap algorithm

As it turns out, even the two-step algorithm allows for further improvements. The swap algorithm can be considered as a sophisticated version of the two-step algorithm. The principal improvement is the fusion of the two steps, streaming and collision, at the implementation level by exploiting only few temporal variables. Hence, unlike in the two-lattice and the shift algorithms, there is no need to allocate additional memory to achieve a fused implementation. The *leitmotif* of the swap algorithm is to break the data dependence between adjacent fluid nodes when traversing the lattice nodes. This is done by explicitly exchanging some of the distribution values of the lattice node at hand with those of neighboring nodes.

To describe the swap algorithm in a compact fashion, we define the sets

$$\mathcal{C}(\vec{d}) = \{a \in [0, b-1] \subset \mathbb{N}\colon (\vec{d} + \vec{c}_a) \in \mathcal{F}\},$$
$$\mathcal{N}(\vec{d}) = \{\vec{l} \in \mathcal{F}\colon \vec{d} + \vec{c}_a = \vec{l},\ a \in \mathcal{C}(\vec{d})\},$$

for every $\vec{d} \in \mathcal{F}_{\mathcal{D}}$. The set $\mathcal{N}(\vec{d})$ includes fluid nodes adjacent to $\vec{d}$ and $\mathcal{C}(\vec{d})$ includes indices of the velocity vectors pointing from $\vec{d}$ to the fluid nodes in $\mathcal{N}(\vec{d})$. Furthermore, we define the sets

$$\mathcal{N}^+(\vec{d}) = \{\vec{l} \in \mathcal{N}(\vec{d})\colon n_f(\vec{d}) < n_f(\vec{l})\},$$
$$\mathcal{C}^+(\vec{d}) = \{a \in \mathcal{C}(\vec{d})\colon n_f(\vec{d}) < n_f(\vec{d} + \vec{c}_a)\}.$$

The set $\mathcal{N}^+(\vec{d})$ includes those fluid nodes in $\mathcal{N}(\vec{d})$ that have a greater enumeration number than $\vec{d}$. Furthermore, the set $\mathcal{C}^+(\vec{d})$ includes indices of the velocity vectors pointing from $\vec{d}$ to the fluid nodes in $\mathcal{N}^+(\vec{d})$. By reversing the inequality signs, analogous definitions are obtained for the sets $\mathcal{N}^-(\vec{d})$ and $\mathcal{C}^-(\vec{d})$.

As the swap algorithm has advanced to a fluid node $\vec{d}$, distribution values $f_a(\vec{d}, t)$, $a \in \mathcal{C}^+(\vec{d})$, are exchanged with distribution values $f_{\bar{a}}(\vec{d} + \vec{c}_a, t)$ of the neighboring nodes in $\mathcal{N}^+(\vec{d})$ (cf. Fig. 4); $\bar{a}$ denotes the parity conjugate of a, i.e. $\vec{c}_a = -\vec{c}_{\bar{a}}$. At the same time, it is assumed that the distribution values $f_{\bar{a}}(\vec{d}, t), \bar{a} \in \mathcal{C}^-(\vec{d})$, have already been exchanged with the distribution values $f_a(\vec{d} + \vec{c}_{\bar{a}}, t)$ of the neighboring nodes in $\mathcal{N}^-(\vec{d})$. Notice that if node $\vec{d} + \vec{c}_i$ is a solid node, the exchange procedure described above automatically takes care of the bounce-back boundary condition. In Fig. 4, the operation of the swap algorithm for the D2Q9 model is illustrated. The fluid nodes in Fig. 4 are assumed to reside well inside the computational domain so that the boundaries of the system do not affect the update procedure.

Immediately after the exchange of distribution values at node $\vec{d}$, i.e. after the streaming step for node $\vec{d}$, the collision step can be performed for the node in question. However, it must be noted that after the streaming step, distribution values $f_i(\vec{d}, t^\star)$ do not reside at the elements of $\vec{V}$ originally assigned for them. Instead, distribution values $f_a(\vec{d}, t^\star)$ and $f_{\bar{a}}(\vec{d}, t^\star)$ associated with velocity vectors $\vec{c}_a$ and $\vec{c}_{\bar{a}}$, respectively, have exchanged their locations in $\vec{V}$ (cf. Fig. 4(b)). This must be taken into account in the collision procedure. During the collision step, updated distribution values $f_i(\vec{d}, t+1)$ are stored to their original locations in $\vec{V}$ (Fig. 4(c)).

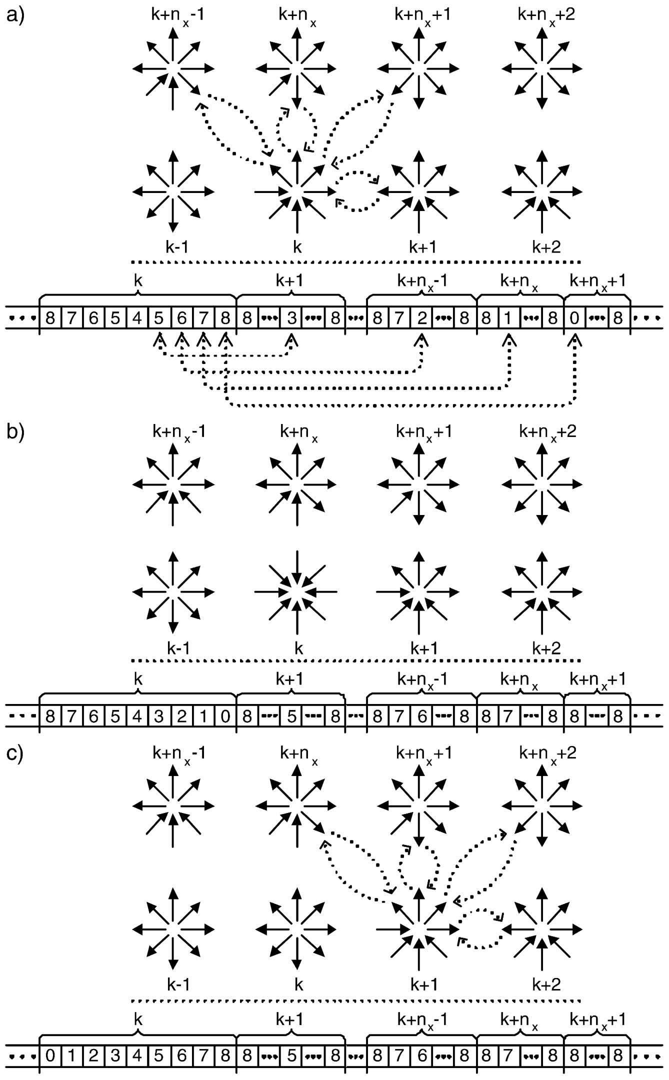

Fig. 4. The swap algorithm for the D2Q9 model has reached a fluid node having enumeration number k. The algorithm advances by (a) streaming into k, (b) colliding at k, and (c) streaming into $k+1$. The contents of the elements in $\vec{V}$ are described with the enumeration numbers of the velocity vectors.

When the swap algorithm reaches node $\vec{d}$, the assumption that the distribution values $f_{\bar{a}}(\vec{d}, t), \bar{a} \in \mathcal{C}^-(\vec{d})$, have already been exchanged with the appropriate distribution values of the neighboring nodes in $\mathcal{N}^-$, is valid for all fluid nodes in $\mathcal{F}_{\mathcal{D}}$ except for those in

$$\mathcal{B}^- = \{\vec{d} \in \mathcal{F}_{\mathcal{D}}\colon \exists a \text{ such that } \vec{d} + \vec{c}_a = \vec{l} \in \mathcal{F}_{\mathcal{G}} \text{ and } n_f(\vec{l}) < n_f(\vec{d})\}.$$

The set $\mathcal{B}^-$ includes all fluid nodes in the domain $\mathcal{D}$, which have neighboring fluid nodes, with a smaller enumeration number, in the ghost layer $\mathcal{G}$. Thus, the nodes in $\mathcal{B}^-$ must be treated separately with a special initialization procedure at the beginning of each time step.

We summarize the swap algorithm in the following way. At the beginning of each time step, the initialization procedure is executed for the nodes in $\mathcal{B}^-$. Thereafter the lattice nodes in the domain $\mathcal{D}$ are iterated starting from the node having the smallest enumeration number up to the node with the largest number.

For each fluid node $\vec{d} \in \mathcal{F}_\mathcal{D}$, the distribution values defined by the index set $\mathcal{C}^+(\vec{d})$ are exchanged with the appropriate distribution values of the neighboring nodes in $\mathcal{N}^+(\vec{d})$. The exchange of the distribution values completes the streaming step for the node $\vec{d}$. The collision step for the node $\vec{d}$ is performed immediately after the streaming step. Notice that in the case of direct or semi-direct addressing, the sets $\mathcal{C}^+(\vec{d})$ and $\mathcal{N}^+(\vec{d})$ need not be truly constructed and stored, but they can be deduced dynamically from the phase vector $\vec{P}$.

As before, let $\{\vec{d}_0, \vec{d}_1, \ldots, \vec{d}_{N-1}\}$ be an ordered set of all lattice nodes in the computational domain $\mathcal{D}$. Guidelines for implementing the swap algorithm can be provided with the following pseudo-code:

```
**for** $t := 0$ **to** $(iters - 1)$ **do**
  enforce_boundary_conditions()
  initialize_swap()
  **for** $k := 0$ **to** $N - 1$ **do**
    **if** element $n(\vec{d}_k)$ of $\vec{P} \geqslant 0$ **then**
      **for** every $a \in \mathcal{C}^+(\vec{d}_k)$ **do** exchange($f_a(\vec{d}_k)$, $f_{\bar{a}}(\vec{d}_k + \vec{c}_a)$)
      collide($\vec{d}_k$)
    **end if**
  **end**
**end**
```

5. Performance

We have simulated three-dimensional Poiseuille flow between two solid plates in order to compare the performances of the four algorithms. The plates are perpendicular to the x-direction and the no-slip condition at the fluid–solid interfaces is realized with the well-known halfway bounce-back boundary condition. The flow is driven in the y direction by a body force, and periodic boundary conditions are applied in the y as well as in the z direction. The LBGK model D3Q19 is utilized; see Section 2 for details. Double precision is used for floating point numbers. The number of fluid nodes between the plates are m_x, m_y and m_z in each coordinate direction. The simulations are run for one hundred time steps. The enumeration function used for the three-dimensional cubic lattice is

$$n(\vec{l}) = l_x \cdot n_y \cdot n_z + l_y \cdot n_z + l_z, \quad \vec{l} \in \mathcal{L}. \tag{7}$$

Simulations were performed on a computer equipped with an AMD Opteron 246 processor and 4 GB memory. The processor has L1 (64 kB) and L2 (1 MB) cache memories for data. The differences in performances between the four algorithms are mainly due to varying degree of cache utilization. The aim of our simulations is to compare performances, and also to demonstrate the effect of memory accessing patterns on the performance. For example, the swap algorithm has a simple memory accessing pattern: when the swap algorithm has advanced to a particular fluid node, there is no need to access memory addresses allocated for distribution values of neighboring nodes having smaller enumeration number than the node at hand. Thus the memory accesses will exhibit more locality, which in turn allows for improved utilization of the cache memories in comparison to the other algorithms.

In order to investigate the interaction between memory accessing patterns and cache memories, we have executed three series of simulations. In each series, two of the domain dimensions are fixed while the dimension in the third direction is varied. In the first simulation series, dimensions $m_y = 40$ and $m_z = 40$ are fixed, and $10 \leqslant m_x \leqslant 200$. The performances of the algorithms are measured in million lattice-site updates per second (MLUPS). The simulations revealed that the variation of m_x has little or no bearing on the performances of the algorithms when m_y and m_z have even moderate values. By simply considering the enumeration function (7), this is no surprise. The difference between the enumeration numbers of two neighboring nodes affects the cache memory utilization. Since this difference does not depend on the dimension of the domain in the x direction, the constant performance in this case is to be expected. In the first simulation series, the approximately constant MLUPS values are 2.47, 2.25, 2.05, and 1.42 for the swap, shift, two-lattice, and two-step algorithms, respectively.

On the other hand, the difference between the enumeration numbers of two neighboring nodes does depend on the dimension in the y direction and even more so on the dimension in the z direction. In the second simulation series, dimensions $m_x = 40$ and $m_z = 40$ are fixed, and $10 \leqslant m_y \leqslant 1000$. With a simplified analysis, performance drops due to the interaction of the memory accessing patterns and the cache memories can be predicted. For example, let us consider the swap algorithm. When the algorithm has advanced to a particular lattice node $\vec{l}$, distribution values from the yz planes $x = l_x$ and $x = (l_x + 1)$ need to be accessed. If the distribution values, and the phase information, for all lattice nodes in these two yz planes fit simultaneously into the L2 cache, good cache memory utilization is probable. One lattice node requires $19 \times 8 + 4$ bytes of memory. Since the dimension in the z direction is fixed in the second simulation series ($n_z = 42$), and the size of the L2 cache is 1 MB, the limiting number of lattice nodes in the y direction can be estimated as follows: $n_y \approx 1\ \mathrm{MB}/(2 \times 42 \times (19 \times 8 + 4)\mathrm{B}) \approx 80$.

When the two-lattice algorithm has advanced to a lattice node $\vec{l}$, distribution values from the yz planes $x = (l_x - 1)$, $x = l_x$ and $x = (l_x + 1)$ need to be read. Furthermore, in the two-lattice algorithm the distribution values are read and stored at different memory addresses (see Section 4). Storing the updated distribution values of every lattice node in the yz plane $x = l_x$ requires approximately $n_y \times n_z \times 19 \times 8$ bytes of memory. Thereby, the estimate for the limiting number of lattice nodes in the y direction is $n_y \approx 1\ \mathrm{MB}/(3 \times 42 \times (19 \times 8 + 4)\mathrm{B} + 42 \times 19 \times 8\mathrm{B}) \approx 40$. Indeed, the second simulation series verifies the performance drops at $m_y \approx 78$ and $m_y \approx 38$ for the swap and two-lattice algorithms, respectively.

Due to limitations in computing resources, the third simulation series is divided into two parts. In the first part, dimensions $m_x = 40$ and $m_y = 40$ are fixed, and $10 \leqslant m_z \leqslant 2000$. In the second part, $m_x = 20$, $m_y = 40$, and $2500 \leqslant m_z \leqslant 8000$. Also the third simulation series verifies the above predicted performance drops (see Fig. 5). According to Fig. 5, all algorithms experience also a second performance drop. In the following analysis, with a z column we refer to the set of lattice nodes having common x and y coordinates. When the swap algorithm

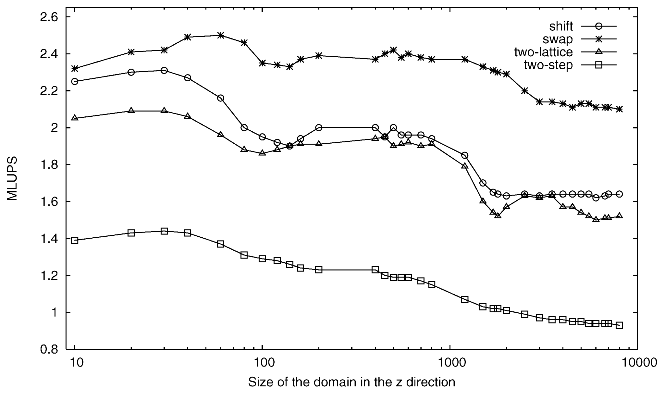

Fig. 5. Performances of the algorithms in the third simulation series, where $m_y = 40$. For $m_z \leqslant 2000$ and for $m_z \geqslant 2500$, dimension in the x direction is $m_x = 40$ and $m_x = 20$, respectively.

has advanced to a particular node, distribution values from five z columns need to be accessed. It is desirable that the distribution values, and the phase information, for all lattice nodes in these five z columns fit simultaneously in the L2 cache. Thus, limiting value for the number of nodes in a z column can be estimated from $n_z \approx 1\ \text{MB}/(5 \times 19 \times 8\text{B} + 5 \times 4\text{B}) \approx 1344$.

To update the distribution values of a particular node in the two-lattice algorithm, distribution values from nine z columns need to be accessed. Storing the updated distribution values of every lattice node in one particular z column requires approximately $n_z \times 19 \times 8$ bytes of memory. Thus, the estimate for the limiting number of lattice nodes in the z direction is $n_z \approx 1\ \text{MB}/(10 \times 19 \times 8\text{B} + 9 \times 4\text{B}) \approx 674$. The estimates $m_z \approx 1342$ and $m_z \approx 672$ for the swap and two-lattice algorithms, respectively, coincide with the performance drops evident in Fig. 5.

The memory requirements of the algorithms are compared by calculating the relative memory consumption: the memory consumptions of the algorithms are scaled by the memory consumption of the swap algorithm. In the first part of the third simulation series, the relative memory consumptions are invariably 1.02, 1.97 and 1.08 for the shift, two-lattice and two-step algorithms, respectively. That is, the two-lattice algorithm consumes almost twice as much memory as the swap algorithm. To summarize, our simulations show that the swap algorithm consistently outperforms the other algorithms, and at the same time the swap algorithm has the lowest memory consumption.

6. Code optimization

The objective of code optimization is to take the general features of modern microprocessor architectures into account when implementing the data structures and algorithms of a program. The goal is to implement the algorithm in such a way that it can be efficiently executed by current microprocessors. In this work, code optimization concentrates on enhancing the memory utilization and the instruction level parallelism of the code.

To enhance memory utilization we try to avoid memory load- and store-operations where possible, and traffic between the main memory and the processor caches, typically the outermost (or slowest) cache level. Standard techniques that can be used to accomplish this include data structures taking advantage of automatic data prefetches and data blocking. To improve instruction level parallelism we rearrange the computations in order to reduce the dependencies between different statements. This makes it possible to utilize the instruction pipeline of the processor more effectively. Here techniques like common subexpression elimination, loop unrolling and branch elimination are utilized. In order to quantify the impact of these techniques on the AMD Opteron processor we have utilized a profiler with hardware performance counters [28] to measure cache hit rates and branch misprediction rates.

The object for our code optimization will be the swap algorithm as presented in Section 4.2. The algorithm is implemented as a parallel program using MPI [29] for message passing. The parallel implementation is based on a straightforward domain decomposition, where the lattice is divided evenly among the processors along the x axis. Each processor updates the fluid nodes in its own subdomain, and uses a layer of ghost nodes to communicate the values at the border between two processors to its nearest neighbors. The performance of the original and the optimized versions were evaluated on three parallel systems.

(A) Pentium III based cluster with 256 kB L2 cache and 100 MB/s switched Ethernet.
(B) Power4 based IBM eServer Cluster 1600 with 1.5 MB L2-cache (shared by two processors) and 512 MB L3 cache (shared by eight processors) and a Federation switch.
(C) AMD Opteron 64 cluster with 1 MB L2 cache and 1 GB/s switched Ethernet.

The measurements were done for a problem size of $100 \times 200 \times 100$ and 100 iterations using 4 processors. The execution times on the Pentium3- and Opteron-based clusters were very stable, varying less than 2% of the total execution time. However, the measurements on the IBM SC showed a very large variation, which could be of up to 20% of the best execution time. We believe this was caused by interference from other programs executing at the same time in the very heavily loaded system.

Most of the execution time, between 74 and 90% depending on the system, is spent in two procedures implementing propagation and relaxation of the distribution values. Therefore, the code optimization efforts concentrate on these two procedures. The interprocess communication accounts for about 25% of the execution time on a slow network (100 MB Ethernet), and 8% on a fast network (IBM SP switch).

The swap algorithm has already by construction taken several aspects of efficient memory utilization into consideration. The data pertaining to a lattice node is placed in consecutive memory locations, thus ensuring that automatic memory prefetches will bring in the data which will be needed next. This is true both for the lattice node being updated as well as for data needed from its lattice neighbors.

6.1. Blocking

Blocking is a standard technique for improving cache utilization (see e.g. [8,9]). The main purpose is to use data already in the caches as efficiently as possible. In the swap algorithm, data pertaining to a particular lattice node will be needed repeatedly. Referring to Fig. 4, in two spatial dimensions, data pertaining to a given lattice node is accessed five times. If the lattice is iterated by blocks of a suitable size instead of row by row, data pertaining to lattice nodes from the next row is still in the cache memory when that row is iterated.

However, special care must be paid to lattice nodes at the border between two blocks. As the swap algorithm has advanced to a fluid node $\vec{d}$, it is assumed that the distribution values $f_{\bar{a}}(\vec{d}, t), \bar{a} \in \mathcal{C}^-(\vec{d})$, have already been exchanged with the distribution values $f_a(\vec{d} + \vec{c}_{\bar{a}}, t)$ of the neighboring nodes in $\mathcal{N}^-(\vec{d})$. If lattice nodes are iterated in the order defined by the enumeration numbers, the assumption is valid for all fluid nodes in $\mathcal{F}_{\mathcal{D}}$ except for those in $\mathcal{B}^-$. If some other iteration order is adopted, say block by block, there are additional nodes at the border between two blocks for which the assumption does not hold. This must be taken into account when implementing blocking for the swap algorithm.

On the Pentium3- and Power4-based systems the performance of the code improved by a factor of 1.1 as a result of the blocking (calculated as execution time of the original program divided by the execution time of the optimized program). We measured the cache hit rates on the Opteron-based system without and with blocking. With blocking we expected to see a clear decrease in the number of cache misses, and this was indeed measured (see Table 2). The block size of course depends on the cache sizes of the processor architecture, and was chosen experimentally to roughly fit within the L2-cache.

6.2. Eliminating dependencies

Modern microprocessors execute instructions out of order. The instruction execution mechanism is allowed to reorder instructions that are independent of each other. Therefore, if unnecessary dependences can be eliminated from the code, the processor has a better opportunity to keep its execution pipeline full.

In the family of LBGK models presented in Ref. [20], the relaxation procedure involves computation of equilibrium distribution functions f_i^{eq}. Due to symmetry, equilibrium distribution functions (3) associated with two opposite velocity vectors have common subexpressions, which should be exploited in order to reduce computation. In the D3Q19 model, there are nine pairs of opposite velocity vectors. Therefore, it is intuitive that the implementation of the relaxation procedure includes nine structurally identical blocks of code, in which the distribution values associated with opposite velocity vectors are updated and restored to their original positions.

Our original implementation used two variables, `common1` and `common2`, to hold common subexpressions exploited in the computation. The same pair of variables were used in all nine code blocks, which introduced a false dependence in the code. By using unique temporary variables `common1`, ..., `common18` for the common subexpressions in each of the nine code blocks, this dependence was eliminated, and the procedure could take better advantage of the out of order execution mechanism. Some additional common subexpression eliminations and simplifications of the procedure were also performed. These optimizations improved the execution time of the code by a factor of 1.07 (compared to the original program) on the Pentium3- and Power4-based systems.

In our profiler setup we found no hardware counter to measure the number of wait cycles when stalling the execution pipeline in the processor and hence we cannot validate our claims for a smaller number of wait cycles via a profiler output. Instead the total execution time is a measure for this, and here we indeed noted the aforementioned speed-up factor of 1.07.

6.3. Eliminating branches

A major cause for breaking the instruction pipeline is the problem of predicting branches correctly. In the swap algorithm with no-slip boundary conditions, fluid lattice nodes that have a solid lattice node as a neighbor, treat their distribution values differently compared to fluid nodes neighboring only other fluid nodes. With deep fluid nodes we refer to fluid nodes surrounded only by other fluid nodes.

For static geometries, instead of checking whether each neighboring node is fluid or solid, we may supplement the program with a table indicating whether a fluid node is a deep fluid node or if it has at least one solid node as a neighbor. This modification eliminates all conditional statements from the code for updating a deep fluid node in the streaming step. The measured effect of the branch elimination was an improvement of about 1.02 times faster than the original program on all architectures.

This statement is supported by our measurements in Table 2, which show that the number of executed branches (retired branches) decreased to about a third. Modern microprocessor architectures already include speculative branch prediction mechanisms which will work very well for simulated systems with high fluid volume fractions, and hence no dramatic improvement occurs in our test cases with essentially only volume boundaries as non-fluid points. For geometries with low fluid fractions or irregular geometries like porous materials we expect the deep fluid mechanism to be more effective. We also noticed that the number of branches clearly increased in program versions with blocking included (e.g., the version with all optimizations). This was to be expected since in blocking the program has to test whether a block boundary has been reached and in that case update the lattice points on the block boundary as described in Section 6.1.

6.4. Optimization results

In addition to the code optimizations described above, some common subexpressions were moved out of loops and functions were declared to be inlined by the compiler. This improved the

Table 1
Measured execution times in seconds and speedup

Optimization	Pentium3	Power4	Opteron
Original version	124.8	39.0	29.3
Optimized version	105.2	28.9	24.7
Speedup	1.19	1.35	1.19

Table 2
Hardware counter measurements using sampling for the original unoptimized swap-program, with one optimization technique included, and the final version with all optimization techniques

	Blocking	Branch elimination
Hardware counter	DATA_CACHE_MISSES	RETIRED_BRANCHES
Original version	57409	127889
One optimization	46796	40990
All optimizations	47597	127903

performance of the program by about 1.1–1.2 times compared to the original version.

The results of the combined code optimization are summarized in Table 1. The overall improvement of the program is a speedup between 1.2 and 1.3, depending on the system. For completeness we have also included in Table 2 the hardware counter results for the final version of the program where all optimizations have been included. As already observed in the previous chapter, optimization techniques often improve on one type of performance but will worsen some other type, e.g., blocking will improve cache performance but increase the number of branches and worsen branch prediction.

Compared to the Pentium3 and the Power4 processors, the Opteron has a considerably more modern design. Both the memory interface (including the cache system) and the instruction execution pipeline (including the branch prediction system) are much more advanced. Therefore, it is not too surprising that the Opteron is a more challenging target for code optimization, and that some optimizations which are efficient on, e.g., the Pentium3 are more or less annihilated by the advanced architectural features of the Opteron.

7. Conclusions

We have investigated algorithms for the implementation of the standard LBE. By simulations it is verified that the two most well-known implementation strategies have their defects: implementations based on the two-lattice algorithm have high memory consumption and implementations realizing the two-step algorithm do not achieve high computational performance. Here a vectorized version of the compressed grid algorithm is considered. For this particular version, the name shift algorithm was proposed. Implementations of the shift algorithm present competitive performance with low memory consumption. Above all, we introduced the swap algorithm, which consistently outperformed other algorithms in our numerical experiments. At the same time the swap algorithm had the lowest memory consumption.

Also, we have demonstrated how performance of LBM implementation can be further improved by common code optimization techniques. Such techniques as blocking, eliminating dependencies, and eliminating branches take advantage of the general features of modern microprocessor architectures. Our code optimization efforts resulted in a speedup by a factor between 1.2 and 1.3, depending on the system.

The shift and swap algorithms can both be implemented with indirect addressing scheme. For example, the streaming step in the swap algorithm corresponds to a permutation of the distribution values. When indirect addressing is applied, this permutation is explicitly stored for every fluid node.

Acknowledgements

The authors would like to thank Professor Jussi Timonen for his contribution during the preparation of this article. Also, the contribution of Dr. Jan Åström at the early stage of this work is gratefully acknowledged.

References

[1] U. Frisch, B. Hasslacher, Y. Pomeau, Lattice-gas automata for the Navier–Stokes equations, Phys. Rev. Lett. 56 (14) (1986) 1505–1508.
[2] G. McNamara, G. Zanetti, Use of the Boltzmann equation to simulate lattice-gas automata, Phys. Rev. Lett. 61 (20) (1988) 2332–2335.
[3] S. Succi, The lattice Boltzmann equation for fluid dynamics and beyond, in: Numerical Mathematics and Scientific Computation, Clarendon Press, Oxford University Press, Oxford Science Publications, New York, 2001.
[4] 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, Comput. & Fluids 35 (8–9) (2006) 888–897.
[5] Z. Guo, T.S. Zhao, Y. Shi, Preconditioned lattice-Boltzmann method for steady flows, Phys. Rev. E 70 (6) (2004) 066706.
[6] O. Filippova, D. Hänel, Grid refinement for lattice-BGK models, J. Comput. Phys. 147 (1) (1998) 219–228.
[7] F. Massaioli, G. Amati, Achieving high performance in a LBM code using OpenMP, in: The Fourth European Workshop on OpenMP (EWOMP 2002), Roma, September 18–20, 2002.
[8] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, U. Rüde, Optimization and profiling of the cache performance of parallel lattice Boltzmann codes, Parallel Process. Lett. 13 (4) (2003) 549–560.
[9] A.C. Velivelli, K.M. Bryden, A cache-efficient implementation of the lattice Boltzmann method for the two-dimensional diffusion equation, Concurrency Comput. Pract. Ex. 16 (14) (2004) 1415–1432.
[10] G. Wellein, T. Zeiser, S. Donath, G. Hager, On the single processor performance of simple lattice Boltzmann kernels, Comput. & Fluids 35 (8–9) (2006) 910–919.
[11] R. Argentini, A.F. Bakker, C.P. Lowe, Efficiently using memory in lattice Boltzmann simulations, Future Gener. Comput. Syst. 20 (6) (2004) 973–980.
[12] N.S. Martys, J.G. Hagedorn, Multiscale modeling of fluid transport in heterogeneous materials using discrete Boltzmann methods, Mater. Struct. 35 (2002) 650–659.
[13] A. Dupuis, B. Chopard, An object oriented approach to lattice gas modeling, Future Gener. Comput. Syst. 16 (5) (2000) 523–532.
[14] M. Schulz, M. Krafczyk, J. Tölke, E. Rank, Parallelization Strategies and Efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers, in: M. Breuer, F. Durst, C. Zenger (Eds.), High-Performance Scientific and Engineering Computing, Proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, March 12–14, 2001, Springer, Berlin, 2002, pp. 115–122.
[15] C. Pan, J.F. Prins, C.T. Miller, A high-performance lattice Boltzmann implementation to model flow in porous media, Comput. Phys. Comm. 158 (2) (2004) 89–105.

[16] T. Abe, Derivation of the lattice Boltzmann method by means of the discrete ordinate method for the Boltzmann equation, J. Comput. Phys. 131 (1) (1997) 241–246.

[17] X. He, L.-S. Luo, A priori derivation of the lattice Boltzmann equation, Phys. Rev. E 55 (6) (1997) R6333–R6336.

[18] X. He, L.-S. Luo, Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation, Phys. Rev. E 56 (6) (1997) 6811–6817.

[19] 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.

[20] Y.H. Qian, D. D'Humières, P. Lallemand, Lattice BGK models for Navier–Stokes equation, Europhys. Lett. 17 (1992) 479–484.

[21] S. Wolfram, Cellular automaton fluids. I. Basic theory, J. Statist. Phys. 45 (3–4) (1986) 471–526.

[22] H. Chen, S. Chen, W.H. Matthaeus, Recovery of the Navier–Stokes equations using a lattice-gas Boltzmann method, Phys. Rev. A 45 (8) (1992) R5339–R5342.

[23] S. Chen, G.D. Doolen, Lattice Boltzmann method for fluid flows, in: Annual Review of Fluid Mechanics, vol. 30, Annual Reviews, Palo Alto, CA, 1998, pp. 329–364.

[24] P. Lavallée, J.P. Boon, A. Noullez, Boundaries in lattice gas flows, Physica D 47 (1–2) (1991) 233–240.

[25] M. Bouzidi, M. Firdaouss, P. Lallemand, Momentum transfer of a Boltzmann-lattice fluid with boundaries, Phys. Fluids 13 (11) (2001) 3452–3459.

[26] A. Koponen, D. Kandhai, E. Hellén, M. Alava, A. Hoekstra, M. Kataja, K. Niskanen, P. Sloot, J. Timonen, Permeability of three-dimensional random fiber webs, Phys. Rev. Lett. 80 (4) (1998) 716–719.

[27] G. Bella, S. Filippone, N. Rossi, S. Ubertini, Using OpenMP on a hydrodynamic lattice-Boltzmann code, in: The Fourth European Workshop on OpenMP (EWOMP 2002), Roma, September 18–20, 2002.

[28] OProfile, URL: http://sourceforge.net/projects/oprofile, 12.9.2006.

[29] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1994.