

# On the Single Processor Performance of Simple Lattice Boltzmann Kernels

G. Wellein<sup>1</sup>, T. Zeiser, G. Hager, S. Donath

*Regionales Rechenzentrum Erlangen, Martensstr. 1, 91058 Erlangen, Germany*

---

## Abstract

This report presents a comprehensive survey of the effect of different data layouts on the single processor performance characteristics for the lattice Boltzmann method both for commodity “off-the-shelf” (COTS) architectures and tailored HPC systems, such as vector computers. We cover modern 64-bit processors ranging from IA32 compatible (Intel Xeon/Nocona, AMD Opteron), superscalar RISC (IBM Power4), IA64 (Intel Itanium2) to classical vector (NEC SX6+) and novel vector (Cray X1) architectures. Combining different data layouts with architecture dependent optimization strategies we demonstrate that the optimal implementation strongly depends on the architecture used. In particular, the correct choice of the data layout could supersede complex cache-blocking techniques in our kernels. Furthermore our results demonstrate that vector systems can outperform COTS architectures by one order of magnitude.

*Key words:* Performance of lattice Boltzmann methods; implementation aspects;

---

## 1 Introduction

Vector computers have long been the architecture of choice for high end applications from computational fluid dynamics (CFD). In the past decade rapid advances in microprocessor technology have led to fundamental changes in high performance computing (HPC). Commodity “off-the-shelf” (COTS) cache-based microprocessors arranged as systems of interconnected symmetric multi-processing (SMP) nodes nowadays dominate the HPC market due to their unmatched price/peak performance ratio. However, it has been acknowledged recently that the gap between sustained and peak performance for scientific applications on COTS platforms is growing continuously [1]. Classical vector systems are presently the only systems which can bridge this

---

<sup>1</sup> Corresponding author: [gerhard.wellein@rrze.uni-erlangen.de](mailto:gerhard.wellein@rrze.uni-erlangen.de)

performance gap especially for memory intensive codes. The demonstration of sustained performance numbers of several TFlop/s by the Earth Simulator (using NEC SX6 vector technology) for a variety of large scale applications [2] has intensified discussions about the relevance of vector computing. In the meantime, a new class of vector processor has been introduced with the Cray X1, which also achieved high sustained performance for a broad range of (vectorizable) applications, according to first user reports [3].

Before putting effort into the parallelization of a code, the serial performance of the implementation should be optimized. Therefore it is the aim of this paper to evaluate characteristics of the single processor performance in order to demonstrate architecture-dependent optimization strategies of a lattice Boltzmann kernel and to comment on the discussion about the relevance of COTS and vector processors for such applications.

We have chosen the lattice Boltzmann method (LBM) [4–8] which has evolved into a promising alternative for the numerical simulation of (time-dependent) incompressible flows during the past decade. The high scientific and commercial potential for large scale applications in many areas [9–12] also calls for a comprehensive performance evaluation of state-of-the art processor architectures. Moreover, the simplicity of the algorithm allows for an easy implementation and comparison of different data layouts as well as optimization approaches.

The remainder of this paper is organized as follows: In Section 2 we first summarize architectural characteristics of the systems and compiler versions used in our study. Section 3 gives a brief introduction to the lattice Boltzmann method. Starting from a naive implementation of the lattice Boltzmann kernel, we point out different architecture-dependent implementation and optimization strategies (Section 4) and discuss the single processor performance together with theoretical estimates for the different machines (Section 5).

Parallelization strategies and parallel performance measurements are discussed in complementary papers [13, 14].

## 2 Architectural Specifications

In Table 1 we briefly sketch the most important single processor specifications of the architectures examined. We focus on processors which are widely used for building large clusters or shared memory nodes as well as on those which have been developed for use in HPC only. Concerning the memory architecture of COTS systems, we find a clear tendency towards on-chip caches which run at processor speed, providing high bandwidth and low latency. The vector

systems (third group of Table 1) incorporate different memory hierarchies and achieve substantially higher single processor peak performance and memory bandwidth. Note that vector systems are much better balanced than COTS systems with respect to the ratio of memory bandwidth to peak performance.

Table 1

Single processor specifications. Peak performance (Peak), maximum bandwidth of the memory interface (MemBW) and sizes of the various cache levels (L1,L2,L3) are given. The L2 (L3) cache of the Cray X1 (IBM Power4) processor are off-chip caches, all other caches are on-chip. The L2 and L3 cache in the IBM p690 is shared by two processors which are placed on a single chip. The Intel Xeon/Nocona processor can be used with a maximum bandwidth of 6.4 GB/s; the benchmark system was equipped with DDR-333 providing a bandwidth of 5.3 GB/s only.

Platform	Single CPU specifications				
	Peak GFlop/s	MemBW GB/s	L1 [kB]	L2 [MB]	L3 [MB]
Intel Xeon DP (3.4 GHz)	6.8	5.3 (6.4)	16	1.0	–
AMD Opteron (1.8 GHz)	3.6	5.3	64	1.0	–
Intel Itanium 2 (1.4 GHz)	5.6	6.4	16	0.25	1.5
IBM Power4 (1.7 GHz)	6.8	9.1	32	1.44	<i>32.0</i>
NEC SX6+ (565 MHz)	9.0	36.0	–	–	–
Cray X1 (800 MHz)	12.8	34.1	–	<i>2.0</i>	–

### Intel Xeon DP and AMD Opteron

The Intel Xeon (codenamed “Nocona” in its current incarnation) and the AMD Opteron processors used in our benchmarks are 64-bit enabled versions of the well-known Intel Xeon/P4 and AMD Athlon designs, respectively, which maintain full IA32 compatibility. Both systems are capable of performing a maximum of two double precision floating point (FP) operations (one multiply and one add) per cycle. The most important difference between both systems is that in standard multi-processor configurations (2- or 4-way systems are in common use) all processors of the Intel based systems have to share one memory bus (reducing the available memory bandwidth per processor) while in AMD based systems the aggregate memory bandwidth scales with processor count.

The AMD Opteron benchmark results presented here have been measured at RRZE on a 4-way Opteron server with an aggregate memory bandwidth of 21.3 GByte/s (=4×5.33 GByte/s). The Portland Group Fortran90 compiler in version 5.2-1 was used.

The Intel Xeon benchmark measurements were done on a 2-way Xeon server with an aggregate memory bandwidth of 5.3 GByte/s (cf. discussion in caption of Table 1). The benchmarks were compiled using the Intel Fortran Compiler for Intel EM64T-based applications in version 8.1.020.

Both systems ran SuSE Linux Enterprise Server 9 for x86\_64 architectures under kernel 2.6.5-7.97-smp.

### **Intel Itanium 2**

The Intel Itanium 2 processor is a superscalar 64-bit CPU using the Explicitly Parallel Instruction Computing (EPIC) paradigm. The Itanium concept does not require any out-of-order execution hardware support but demands high quality compilers to identify instruction level parallelism at compile time. Today clock frequencies of up to 1.6 GHz and on-chip L3 cache sizes from 1.5 to 6 MB are available. Two Multiply-Add units are fed by a large set of 128 FP registers, which is another important difference to standard microprocessors with typically 32 FP registers. The basic building blocks of systems typically used in scientific computing are two-way nodes (e.g. SGI Altix, HP rx2600) sharing one bus with 6.4 GByte/s memory bandwidth.

The system of choice in our report is a two-way HP zx6000 workstation with 1.4 GHz/1.5 MB CPUs running SuSE Linux Enterprise Server 9 with SMP kernel 2.6.5-7.97. Unless otherwise noted, the Intel Fortran Itanium Compiler V8.0 (8.0.046\_pl050.1) was used.

### **IBM Power4**

The IBM Power4 is a 64-bit superscalar out-of-order RISC processor with a maximum clock frequency of 1.7 GHz and two Multiply-Add units. If used in the IBM pSeries 690, four chips (eight processors) are placed on a Multi-Chip-Module (MCM) and can use a large interleaved (external) L3 cache of 128 MB aggregated size.

The Power4 measurements reported in this paper were done on a single IBM p690 node (1.7 GHz Power4+) at NIC Jülich with the bus frequency being  $\frac{1}{3}$  of the core frequency.

### **NEC SX6+**

From a programmers' view the NEC SX6+ is a traditional vector processor with 8-track vector pipes running at 565 MHz. One multiply and one add instruction per cycle can be executed by the arithmetic pipes delivering a peak performance of 9 GFlop/s. The memory bandwidth of 36 GByte/s allows for one load or store per Multiply-Add instruction. The processor contains 64 vector registers, each holding 256 64-bit words. Vectorization of the application code is a must on this system. An SMP node comprises eight processors and provides a total memory bandwidth of 256 GByte/s, i. e. the aggregated single processor bandwidths can be saturated.

The benchmark results presented in this paper were measured on a NEC SX6+ at the High Performance Computing Center Stuttgart.

## Cray X1

The basic building block of the CRAY X1 is a *multi-streaming processor* (MSP) which one usually refers to as processor or CPU. The MSP itself comprises four processor chips (single-streaming processor [SSP]), each incorporating a superscalar and a vector section. The vector section contains 32 vector registers of 64 elements each and a two-pipe processor capable of executing four double precision FP operations and two memory operations. Running at a clock speed of 800 MHz, one MSP can perform up to 16 double precision FP operations per cycle (12.8 GFlop/s) and its memory interface provides 34.1 GByte/s bandwidth. In contrast to classical vector processors, the vector units can use an L2 cache in situations where cache blocking can be applied. At first glance long vectorized loops are the preferred programming style since the MSP unit can operate in a way similar to classical wide-pipe vector processors such as the NEC SX6. However, it is also possible to parallelize a loop nest on an outer level using the SSPs, thereby reducing the impact of vector start-up times.

The benchmark results presented in this paper have been provided by Cray.

## 3 Basics of the Lattice Boltzmann Method

The widely used class of lattice Boltzmann models with BGK approximation of the collision process [4–7] is based on the evolution equation

$$f_i(\vec{x} + \vec{e}_i \delta t, t + \delta t) = f_i(\vec{x}, t) - \frac{1}{\tau} [f_i(\vec{x}, t) - f_i^{\text{eq}}(\rho, \vec{u})] \quad i = 0 \dots N. \quad (1)$$

Here,  $f_i$  denotes the particle distribution function which represents the fraction of particles located in timestep  $t$  at position  $\vec{x}$  and moving with the microscopic velocity  $\vec{e}_i$ . The relaxation time  $\tau$  determines the rate of approach to local equilibrium and is related to the kinematic viscosity of the fluid. The equilibrium state  $f_i^{\text{eq}}$  itself is a low Mach number approximation of the Maxwell-Boltzmann equilibrium distribution function. It depends only on the macroscopic values of the fluid density  $\rho$  and the flow velocity  $\vec{u}$ . Both can be easily obtained as the first moments of the particle distribution function.

The discrete velocity vectors  $\vec{e}_i$  arise from the  $N$  chosen collocation points of the velocity-discrete Boltzmann equation and determine the basic structure of the numerical grid. A typical discretization in 3-D is the D3Q19 model [7] which uses 19 discrete velocities (collocation points). It results in a computational domain with equidistant Cartesian cells (voxels) as shown in Fig. 1.

Each timestep ( $t \rightarrow t + \delta t$ ) consists of the following steps which are repeated for all cells:

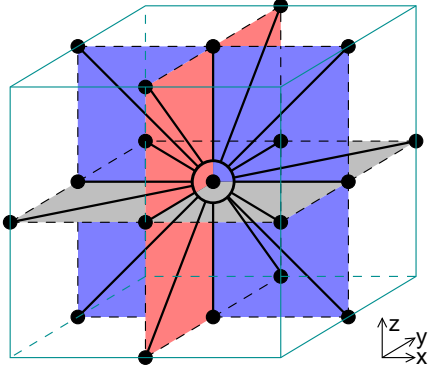


Fig. 1. Discrete velocity vectors in the different planes for the D3Q19 LBM model.

- Calculation of the local macroscopic flow quantities  $\rho$  and  $\vec{u}$  from the distribution functions,  $\rho = \sum_{i=0}^N f_i$  and  $\vec{u} = \frac{1}{\rho} \sum_{i=0}^N f_i \vec{e}_i$ .
- Calculation of the equilibrium distribution  $f_i^{\text{eq}}$  from the macroscopic flow quantities (see [7] for the equation and parameters) and execution of the “collision” (relaxation) process,  $f_i^*(\vec{x}, t^*) = f_i(\vec{x}, t) - \frac{1}{\tau} [f_i(\vec{x}, t) - f_i^{\text{eq}}(\rho, \vec{u})]$ , where the superscript \* denotes the post-collision state.
- “Propagation” of the  $i = 0 \dots N$  post-collision states  $f_i^*(\vec{x}, t^*)$  to the appropriate neighboring cells according to the direction of  $\vec{e}_i$ , resulting in  $f_i(\vec{x} + \vec{e}_i \delta t, t + \delta t)$ , i.e. the values of the next timestep.

The first two steps are computationally intensive but involve only values of the local node while the third step is just a direction-dependent uniform shift of data in memory. A fourth step, the so called “bounce back” rule [15], is incorporated as an additional part of the propagation step and “reflects” the distribution functions at the interface between fluid and solid cells, resulting in an approximate no-slip boundary condition at walls.

## 4 Implementation and Optimization Strategies

The explicit lattice Boltzmann algorithm as outlined above can be easily implemented. By removing contributions from zero components of  $\vec{e}_i$  as well as precomputing common subexpressions, the number of floating point operations (Flops) to be executed can be significantly reduced. This results in roughly 200 Flops per cell update for the D3Q19 BGK model. The actual number can vary because of compiler optimizations.

A straightforward implementation could consist of three nested loops over the three spatial dimensions and treat the collision step independently from the propagation process: First, the values of the current timestep are read from the local cell, then the relaxation is executed and the results are written back to a temporary array which is only available within the collision-propagation routine. This can be done independently for all cells. Finally, in a separate

propagation loop, these values are loaded again and written back to adjacent cells of the original, globally available array. In order to be able to do the propagation uniformly even at the boundaries of the computational domain, an additional ghost layer is added around the actual domain of interest thus increasing the array size by one in each direction.

Improved implementations, in particular concerning data transfer and access, are discussed in more detail in the following subsections.

#### 4.1 Standard Version

The number of data transfers can be reduced by executing the collision process and propagation step in one loop. Using two arrays — holding data of successive timesteps  $t$  and  $t+1$  — keeps the implementation simple and avoids data dependencies between the two timesteps. During an update, values are read from one array and written to the other including the propagation step (within the reading or writing step). At the end of each timestep the two arrays are switched, i.e. the source becomes the destination and vice-versa. Depending on the implementation, the propagation step is realized as first or last step of the iteration loop, resulting in a “pull” or “push” scheme of the update process:

<p><b>Pull:</b></p> <ul style="list-style-type: none"> <li>• read distribution functions from <i>adjacent</i> cells, i.e. <math>f_i^*(\vec{x} - \vec{e}_i \delta t, t - \delta t)</math></li> <li>• calculate <math>\rho</math>, <math>\vec{u}</math> and <math>f_i^{\text{eq}}</math></li> <li>• write updated values to <i>current</i> cell, i.e. <math>f_i^*(\vec{x}, t)</math></li> </ul>	<p><b>Push:</b></p> <ul style="list-style-type: none"> <li>• read distribution functions from <i>current</i> cell, i.e. <math>f_i(\vec{x}, t)</math></li> <li>• calculate <math>\rho</math>, <math>\vec{u}</math> and <math>f_i^{\text{eq}}</math></li> <li>• write updated values to <i>adjacent</i> cells, i.e. <math>f_i(\vec{x} + \vec{e}_i \delta t, t + \delta t)</math></li> </ul>
---	---

Obviously, the main difference consists in non-local read operations (gather data) in the first case compared to non-local write operations (scatter data) in the second.

For the remainder of the paper, the “push” scheme is used. Collision and propagation are done in one step and the values of the distribution functions are stored in one array of dimension 5, indexed by the three spatial coordinates, the discrete-velocity direction  $i$  and two timesteps ( $t$  or  $t^*$ ). In Fig. 2 a sketch of this *standard version* is given. The order of the indices of  $f$  determines the memory access pattern and has substantial performance impact. The index orders  $(i, x, y, z, t)$  and  $(x, y, z, i, t)$  have been considered in this report with the first index addressing consecutive memory locations due to Fortran’s column major order.

Fig. 2. Code snippet of *standard version* with *PropOpt* data layout.

```

real*8 f(0:Nx+1,0:Ny+1,0:Nz+1,0:18,0:1)
logical fluidCell(1:Nx,1:Ny,1:Nz)
real*8 dens, ne, ...
do z=1,Nz; do y=1,Ny; do x=1,Nx
  if ( fluidCell(x,y,z) ) then
    ! read distributions from local cell
    ! and calculate moments
    dens=f(x,y,z,0,t)+f(x,y,z,1,t)+ &
      f(x,y,z,2,t)+...
    ...
    ! compute non-equilibrium parts
    ne0=...
    ...
    ! write updates to neighboring cells
    f(x ,y ,z , 0,tN)=f(x,y,z, 0,t)*....
    f(x+1,y+1,z , 1,tN)=f(x,y,z, 1,t)*....
    ...
    f(x ,y-1,z-1,18,tN)=f(x,y,z,18,t)*....
  endif
enddo; enddo; enddo

```

Fig. 3. Code snippet of *CollOpt-3D* with blocked loops.

```

real*8 f(0:Nx+1,0:Ny+1,0:Nz+1,0:18,0:1)
logical fluidCell(1:Nx,1:Ny,1:Nz)
real*8 dens, ne, ...
! outer loops with increment blksize
do zz=1,Nz,blksize
do yy=1,Ny,blksize
do xx=1,Nx,blksize
  ! inner loops with length <= blksize
  do z=zz,min(Nz, zz+blksize-1)
  do y=yy,min(Ny, yy+blksize-1)
  do x=xx,min(Nx, xx+blksize-1)
    ! no further modifications required here
    ! as x,y,z get the same values as in the
    ! original algorithm
    if ( fluidCell(x,y,z) ) then
      ... same computations as in the
      algorithm on the left
    endif
  enddo; enddo; enddo
enddo; enddo; enddo

```

## 4.2 Data Layout and Cache Optimization

On cache-based machines, main memory bandwidth and latencies are serious bottlenecks for memory intensive applications like LBM. In particular, the organization of data transfer in units of cache lines (typically 64 or 128 bytes) often results in severe performance problems for non-consecutive data access. In our implementation each element of the distribution function  $f$  is accessed only once in each timestep. Thus, the major aim of our optimizations for cache-based processors should be to exploit all entries of a cache line once it is resident in any level of the cache hierarchy. Using the *standard version* described above, we first consider the impact of the data layout. For the sake of simplicity we assume a cache line length of 16 double words and take into account one cache level only. We also do not comment on the effect of the *if* clause since we focus on problems with low obstacle-densities such as channel flows. In this section we use the Intel Xeon processor to substantiate the effects of different implementations and data layouts.

### Collision Optimized Data Layout (CollOpt)

In the *collision optimized* data layout, the 19 distributions of the current cell,  $f(0:18, x, y, z, t)$ , are located contiguously in memory. This is conventionally called an “array-of-structures” arrangement. Two cache lines are loaded for the collision process, if none of the distributions are available in cache. The first entries of these two cache lines are used for the computations of the current cell, the remaining ones will be used in the immediately following next inner loop iteration where  $f(0:12, x+1, y, z, t)$  are used directly from the cache. Thus, all entries from the loaded cache lines are used. In the propagation step, however, the 19 results have to be stored to non-contiguous memory locations. Many different cache lines (in worst case 19) are involved but only a few entries

(1–3) of each line actually get modified at the same time. Furthermore, the distributions of a certain target cell,  $f(0:18, x, y, z, t + 1)$ , are updated by neighbors from three different  $z$ -planes ( $z - 1, z, z + 1$ ).

Between all these updates, three complete  $z$  planes of the two distribution functions ( $t$  and  $t + 1$ ) with a total size of

$$\begin{aligned} \text{Mem}_Z &= 2 \times 19 \times (Nx + 2) \times (Ny + 2) \times 3 \times 8 \text{ Byte} \\ &\approx (Nx + 2) \times (Ny + 2) \times 1 \text{ KByte} \end{aligned} \quad (2)$$

are loaded from main memory to cache. If the cache capacity is less than  $\text{Mem}_Z$  it is likely that the cache lines holding  $f(0:18, x, y, z, t + 1)$  are replaced by other data before all the entries have been updated and therefore must be reloaded several times. Consequently, we find a significant drop in performance for the *collision optimized* data layout on Intel Xeon (1 MByte L2 cache) at  $Nx, Ny > 32$  (cf. Fig. 4), where  $\text{Mem}_Z > 1156 \text{ KByte}$  holds.

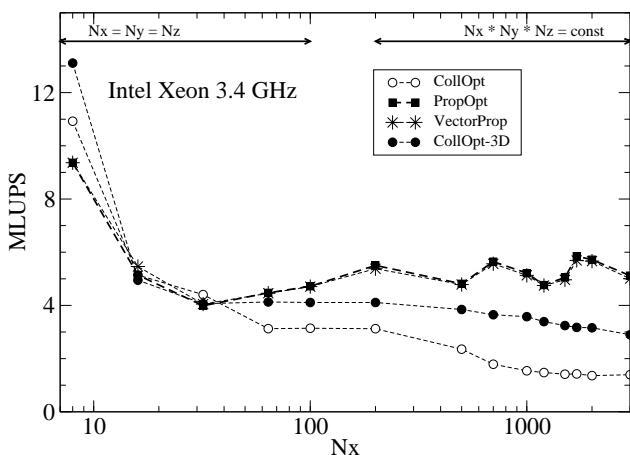


Fig. 4. Lattice site update rate (cf. Section 5) for Intel Xeon as function of the inner loop length  $Nx$ . For  $Nx \leq 100$  cubic geometries have been chosen, while for  $Nx \geq 200$  the total domain size has been fixed starting with  $(Nx = 200; Ny = Nz = 70)$ . For *CollOpt-3D* which uses cubic blocking combined with *collision optimized* data layout a block size of  $\overline{Nx} = 8$  was chosen.

If  $Nx$  is further increased a second performance drop is visible at  $Nx \approx 500 - 1000$ . Following the discussion above, at that point the L2 cache cannot even hold all data required to compute three successive lines of dimension  $Nx$  in a fixed  $z$  plane:  $\text{Mem}_Y \approx (Nx + 2) \times 1 \text{ KByte}$

The combination of *standard implementation* and *collision optimized* data layout is referred to as *CollOpt* in what follows.

### Propagation Optimized Data Layout (PropOpt)

The problems of the *collision optimized* data layout can easily be resolved by using the *propagation optimized* data layout ( $f(x, y, z, 0:18, t)$ ), conventionally called the “structure-of-arrays” arrangement. The inner loop index  $x$  now addresses successive memory cells. In the propagation step, the updates are

done separately for each direction but consecutive in memory regarding the inner loop index. Once the 19 discrete velocity components have been loaded to the cache (cf. discussion about *write allocate* strategy in sect. 5.1), they can be modified in-cache on successive  $x$  iterations. The same holds for the collision where at first 19 different cache lines are loaded. Afterwards, the cached data can be accessed quickly in the successive  $x$  iterations. Of course this layout only pays off because typical cache sizes ( $\sim 1$  MByte) are much larger than the aggregate size of the cache lines accessed separately in collision and propagation steps:  $2 \times 19 \times 16 \times 8 \text{ Byte} \approx 5 \text{ KByte}$ .

The combination of *standard implementation* and *propagation optimized* data layout is referred to as *PropOpt* in what follows.

A well-known pitfall of the *PropOpt* data layout are severe cache trashing effects if powers of two are used in the declaration of the spatial array dimensions for the distribution function. E.g. declaring *real\*8 f(0:127, 0:127, 0:127, 0:18, 0:1)* and assuming an 8-way associative cache of size 1 MB (as implemented in the L2 cache of Intel Xeon) at least all the  $19 \times 2$  components of both distribution functions at a fixed lattice cell are mapped to the same 8 cache locations forcing associativity conflicts and frequent cache line replacements. Fig. 5 clearly demonstrates the effect, which is more pronounced at larger  $Nx$  due to the L2 cache organization. The problem of cache trashing can be avoided by array

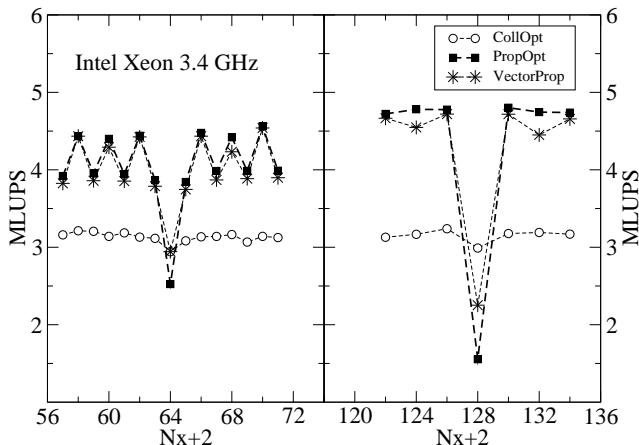


Fig. 5. Cache trashing effect for Intel Xeon at cubic ( $Nx = Ny = Nz$ ) computational grid sizes close to  $Nx = 64$  and  $Nx = 128$ . The  $x$ -axis represents the size of the array declaration for the distribution function including 2 additional ghost layer in each spatial dimension

padding.

It is obvious that cache trashing is not a problem for the *collision optimized* data layout because the leading array dimension is not a power of two but the number of discrete velocity directions (which is 19 in our case).

In very good agreement with above considerations about data access patterns, Figs. 4 and 5 clearly demonstrate the benefit of the *PropOpt* version compared to the *CollOpt* implementation.

### 3D Blocking (CollOpt-3D)

A familiar technique to improve the locality of data access, especially for the *CollOpt* implementation, consists in blocking the three spatial dimensions and working in the three innermost loops on very small cubic grids only. Replacing

the actual dimensions of the computational grids( $Nx, Ny, Nz$ ) in Fig. 2 by small domain dimensions  $\overline{Nx}, \overline{Ny}, \overline{Nz}$  (e.g.  $\overline{Nx} = \overline{Ny} = \overline{Nz} = 8$ ), requires three additional outer loops which provide the offsets of the small domain with respect to the whole computational domain (see Fig. 3). For a more detailed discussion we refer to Ref. [16]. If using 3D Blocking we can replace the sizes of the computational grid in Equation 3 by the sizes of the small inner block and find  $\text{Mem}_z \approx 100$  KByte for  $\overline{Nx} = \overline{Ny} = 8$ . In this case, at least the three  $z$ -planes involved in the propagation step for the small domain fit into the L2 cache of Xeon processors. Consequently, the performance drop of *CollOpt* version for  $Nx > 32$  is suppressed when using 3D blocking as depicted in Fig. 4 by the measurements denoted as *CollOpt-3D*. The slow decline in performance of *CollOpt-3D* at very large  $Nx$  can be attributed to the fact that cubic blocking is used even for extremely long but rather thin channels (e.g.  $Nx = 3000$ ;  $Ny = Nz = 18$ ) which causes some overhead. Most notably, for intermediate to large problem sizes the *CollOpt-3D* is still significantly slower than *PropOpt* version, which could be caused by repeated loading of cache lines which involve fluid cells at the surface of the small domains. In principle, this effect could be minimized by increasing the block size. This is however not practical because  $\text{Mem}_z$  would soon exceed the cache size.

### 4.3 Vector Version

For vector architectures, long inner vectorizable loops are mandatory. In our *vector version* the three nested spatial loops ( $x, y, z$ ) are fused into just one large loop ( $m$ ) which can be fully vectorized. The distribution function  $f$  is redefined to a three-dimensional array with the first index running over all cells (e.g.  $f(x, y, z, 0:18, t) \rightarrow f(m, 0:18, t)$  with  $m = 0, \dots, (Nx + 1) * (Ny + 1) * (Nz + 1)$ ). An appropriate mask blocks out the ghost cells (i.e. cells with  $x = 0$ ,  $x = Nx + 1$ , etc.) from the calculation and propagation process. Throughout this report we mainly use the *propagation optimized* data layout for the vector version and denote these measurements as *VectorProp*. In Figs. 4 and 5 we find that the performance of the *VectorProp* implementation is almost identical with the *PropOpt* results. This is not a big surprise since both implementations use the same data layout and have comparable data access patterns. The only difference is that the vector loop index also runs over the additional ghost layers, which however are blocked from computations by the mask ( $fluidCells(x, y, z)$ ).

#### 4.4 Cray X1 Cache Bypass Strategy

The L2 cache of the Cray X1 provides a mechanism for efficiently executing short vector loops. Contrary to typical cache implementations, the usage of the L2 cache can selectively be controlled on an array level by a compiler directive (`!DIR$ NO_CACHE_ALLOC`). So-called non-allocating loads or stores will then bypass the cache. In order to be able to tune read and write operations to the distribution function at the two different time levels separately, we use two arrays ( $f_t$  and  $f_{t+1}$ ) of dimension 4 for the CRAY X1 measurements.

#### 4.5 Other Optimization Strategies

In the literature one can find other recent optimization strategies which aim at reducing memory consumption and/or improving performance.

Pohl *et al.* [16] extended the idea of blocking and demonstrate the effect of n-way blocking. In particular a 4-way blocking (3 fold spatial blocking and additional blocking in time) can provide additional performance improvements on some architectures.

Pohl *et al.* [16] and Schulz *et al.* [17] presented two different “compressed grid” approaches, i.e. they reduce the total memory consumption almost by a factor of two by allocating only memory for one set of distribution functions on a slightly enlarged domain. Already used distribution values of the current timestep are replaced on the fly by values of the new timestep during the propagation step, carefully obeying data dependencies.

However, the 4-way blocking algorithm as well as the compressed grids make it much more difficult to incorporate advanced boundary conditions or models for more complicated physics as both might depend on pre- as well as post-collision values of more than just the local cell itself. Therefore, they will not be further investigated in this paper.

Argentini *et al.* [18] use the non-BGK model of Ladd [19] and thus succeed in storing only 9 moments of the distribution functions instead of all the distribution values themselves. However, they admit that an application of this idea to the common BGK model is not (transparently) possible.

Pan *et al.* [20] as well as Schulz *et al.* [17] presented data structures — in particular for porous media applications with low porosities (i.e. with a low ratio of fluid to solid nodes) — which abandon the “full matrix representation”. Instead, they use 1-D lists with the data of the distribution functions and the connectivity. This eliminates the need for storing bulk obstacle cells. For low porosities, a lot of memory can be saved in this way. However, memory access patterns get much more complicated and include massive indirect addressing.

As long as vectorization is not prevented, vector systems probably can do rather well despite the indirect memory access [17]. However, on RISC machines, cache reuse is significantly reduced by the scattered memory access and thus a considerable performance loss is expected. Careful data ordering in the lists (e.g. Morton ordering as shown in [20]) can lessen the performance impact to some extent. However, as the optimal reordering process is probably np-complete, some heuristics will always be required.

## 5 Results

In the following, an estimation of theoretically achievable performance numbers on different platforms as well as performance measurements with the different implementations and optimization strategies as outlined above are discussed. All performance numbers are given in MLUPS (**M**ega **L**attice **S**ite **U**pdates **p**er **S**econd), which is a handy unit for measuring the performance of LBM. It allows an easy estimation of the runtime of a real application depending on the domain size and the number of desired timesteps only.

### 5.1 Performance Estimation

Based on characteristic quantities of the benchmarked architectures (cf. Table 1), an estimate for a theoretical performance limit can be given. Performance is either limited by available memory bandwidth or peak performance. Thus, the attainable maximum performance  $P$  in MLUPS is either given as

$$P = \frac{\text{MemBW}}{B} \quad \text{or} \quad P = \frac{\text{Peak Perf.}}{F} \quad (3)$$

where  $B$  is the number of bytes per cell to be transferred from/to main memory and  $F$  is the number of floating point operations per cell. Considering the memory bandwidth as the limiting factor and assuming a *write allocate* strategy (additional cache line load is performed on a write miss), we find  $B = 3 \times 19 \times 8$  bytes = 456 bytes (per cell for the D3Q19 model). Without the write allocate requirement (e.g. on NEC SX6), only 304 bytes have to be transferred. While the memory bandwidth is given by the architecture, the average number of FP operations per cell slightly depends on processor details and compiler. Following the discussion in Section 4 we choose  $F = 200$  as a reasonable guess for the number of FP operations per cell, which yields a performance equivalent of 1 GFlop/s  $\approx$  5 MLUPS. Table 2 shows the estimated performance numbers and the maximum of the measured update rates for the D3Q19 model with a domain size of  $128^3$  for the benchmarked architectures

Table 2

Maximum theoretical performance in MLUPS if limited by peak performance (Peak) or memory bandwidth (MemBW). The last column presents the maximum performance measured for a domain size of  $128^3$ . For the CRAY X1 measurement presented, the L2 cache was bypassed.

Platform	max. MLUPS		measurement	
	Peak	MemBW	MLUPS	
Intel Xeon (3.4 GHz)	34.0	11.8	4.8	PropOpt
AMD Opteron (1.8 GHz)	18.0	11.8	2.7	PropOpt
Intel Itanium 2 (1.4 GHz)	27.0	14.0	7.6	PropOpt
IBM Power4 (1.7 GHz)	34.0	20.0	5.9	VectorCol
NEC SX6+ (565 MHz)	45.0	118	41.3	VectorProp
Cray X1 (1 MSP)	64.0	112	34.9	VectorProp

using the single processor peak performance and memory bandwidth numbers given in Table 1.

Interestingly, on the vector system, in particular on the NEC SX6+, the performance of our kernel is no longer limited by the memory bandwidth.

## 5.2 Performance evaluation

The results presented are averages of several benchmark runs on each machine. In particular, for AMD Opteron and IBM Power4 measurements we found a significant variation ( $\approx 10\%$ ) in performance for identical runs. Concerning the system sizes we have chosen cubic systems up to  $Nx = 128$ . For larger values of  $Nx$ , the total system size has been fixed, i.e. we use long tubes with increasing length and decreasing cross-sections (cf. also discussion in caption of Fig. 4).

### 5.2.1 Performance of Intel Xeon and AMD Opteron

The performance data for Intel Xeon were already presented in detail in Figs. 4 and 5 and we mainly focus here on the AMD Opteron. For all domain sizes, the AMD Opteron processor is significantly behind Intel Xeon (Fig. 6). More importantly, the performance shortfall in our measurements is much larger than expected based on the lower peak performance (for small data sets) and lower memory bandwidth (intermediate and large data sets) of the processor itself. For a  $128^3$  lattice we find that while the Xeon processor gets 41% of achievable performance (based on a memory bandwidth of 5.3 GByte/s; cf. discussion in caption of Table 1), the Opteron is only at 23%. Considering that Opteron processors have an integrated on-chip memory controller, they should sustain at least the same fraction of theoretical performance as the Xeons. One

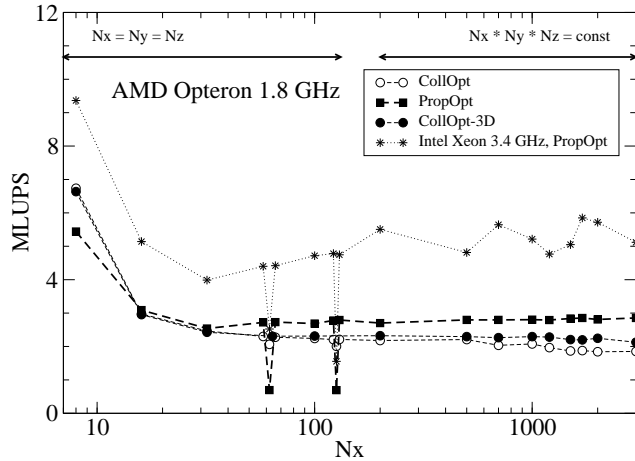


Fig. 6. Lattice site update rate for AMD Opteron. For comparison *PropOpt* performance numbers of the Intel Xeon processor are included.

could speculate that the Portland Group compiler, although developed for the AMD Opteron, still generates sub-optimal code. As a second indication for this assumption we find only a small performance variation for the different memory layouts and only a minor performance increase through 3D blocking for the Opteron processor. With advances in PGI's compiler technology, one might expect those shortcomings to be remedied in the future. Finally, it must be emphasized that the scalability of memory bandwidth in Opteron based multiprocessor systems — which is a strength of the AMD design — has not been tested in our measurements. It was demonstrated for a closely related LBM application in Ref. [13] that the performance gap with respect to Intel designs can be substantially reduced when using both processors of a 2-way node.

### 5.2.2 Performance of IBM Power4+

The performance characteristics obtained on one CPU of an IBM Power4+ p690 node (cf. Fig. 7 and Table 2) are vastly different from the results of Intel/AMD systems and do not fit into our considerations concerning optimal data layout. In contrast to all cache-based microprocessor covered in our

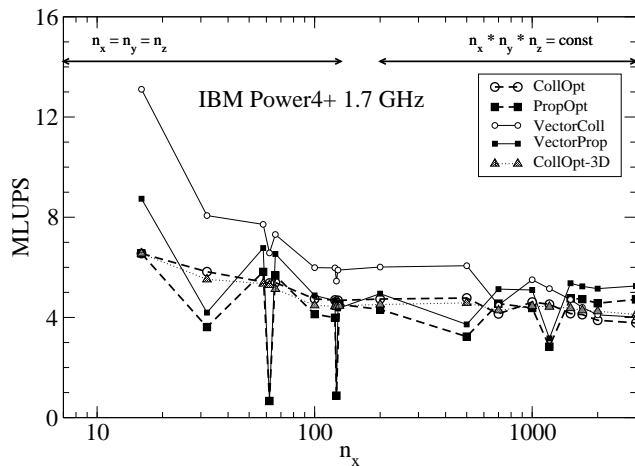


Fig. 7. Lattice site update rate of a single IBM Power4+ CPU on a fully equipped IBM p690 node.

report, the *propagation optimized* data layout has not proven to be optimal for the IBM architecture. For most system sizes, the *vector* implementation (one long loop) combined with *collision optimized* data layout shows a significant improvement in comparison to the *standard* version with three nested loops. Obviously, the Power4+ has a quite large overhead for starting loops or the compiler has difficulties in optimizing the loop nest with the large body on the innermost level. A second indication for this assumption is that 3-D blocking (*CollOpt-3D*) does not perform better than *CollOpt*. A clear benefit from the large aggregated L3 cache size of a p690 MCM can also not be seen.

Although the Power4+ CPU is roughly 20% ahead of the Intel Xeon, the fraction of sustained versus theoretical performance is less than 30%. However, the high variation in the measurements (even if using a dedicated node) and the strange performance characteristics indicate that there is room left for performance improvements through an optimization of system parameters and advances in compiler technology, respectively.

### 5.2.3 Performance of Intel Itanium 2

At first glance, the Intel Itanium 2 processor shows similar characteristics as IA32 compatible processors concerning optimal data layout and cache effects (see Fig. 8). While the cache trashing effects again occur at same  $N_x$  values,

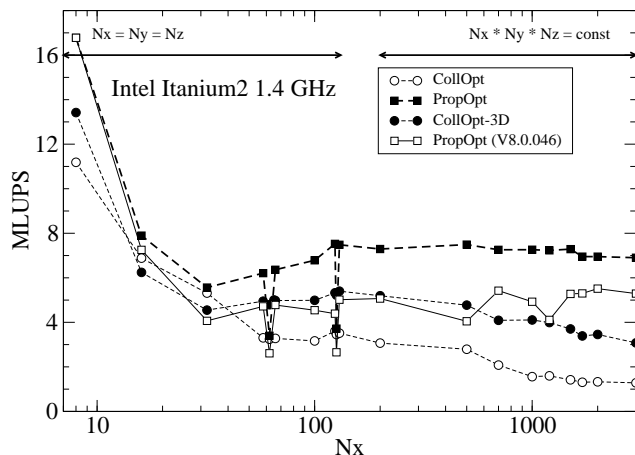


Fig. 8. Lattice site update rate for Intel Itanium 2 processor. Besides performance numbers using the default compiler (V8.0.046\_pl050.1; Build 20040416; see Section 2) the *PropOpt* performance using the first release of the same compiler version (V8.0.046; Build 20040416) is depicted.

the second drop in performance of *CollOpt* is shifted to higher  $N_x$  values because of a 50% larger cache capacity.

The main difference to all other COTS processors is that the Itanium processor provides a significantly higher performance level for all domain sizes and achieves roughly 50% of the maximum achievable performance given in Table 2. In particular, if we consider that the single processor specifications of all COTS systems are comparable (see Table 1) the Itanium architecture seems to be extremely well suited for the numerical requirements of LBM. Although Intel Xeon and IBM Power4 processor provide 20% higher peak performance they are 30% to 50% lower in performance even if the total computational

domain ( $8^3$  or  $16^3$ ) fits into on-chip caches. In this context, the large register set of Itanium 2 reduces register spills which cause additional traffic between caches and registers and arise due to the very complex loop body of LBM. However, also for intermediate and large data sets where memory access is the limiting factor Itanium 2 is ahead of its competitors. Here the large number of prefetch instructions which can be handled both by compiler and memory subsystem helps to make best use of the memory interface. According to the compiler logfile, one prefetch instruction for each of the 38 velocity components of the *PropOpt* distribution functions is issued.

A well-known problem of Itanium performance is the quality of the compiler. To demonstrate the sensitivity to even minor changes in the same compiler version, performance results for the original build of our default compiler were included in Fig. 8. Only the in-cache performance is the same for both versions. For all other system sizes the early build generates code that is roughly 30% slower than when using the more current compiler build (issued 3 months later). Comparing the logfiles of both versions reveals that the prefetching is done in a completely different way.

Improvements in compiler technology could be one reason why Itanium performance is almost a factor of two better than the results reported in Ref. [16], where compiler version 7.0 was used.

#### 5.2.4 Performance of Vector Processors: NEC SX6+ and Cray X1

The remarkable and distinguishable observation for vector systems is that performance does not decrease even for very large domain sizes (Fig. 9). On the contrary, performance saturates at a high level with increasing problem size. At first sight, one might attribute the low performance with small problem sizes to the overhead of loading short vectors when dealing with small domains. Detailed profiling of the code on both vector systems showed, however,

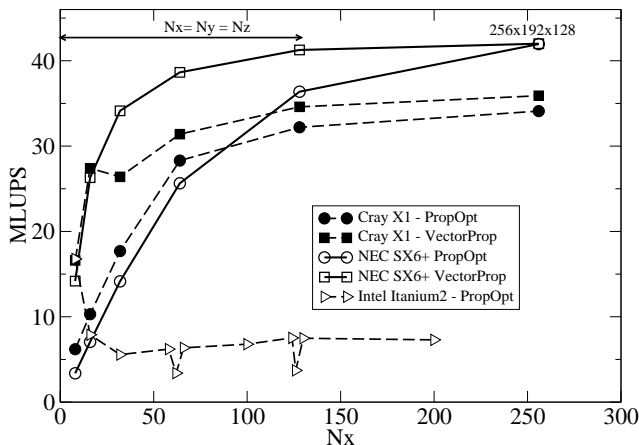


Fig. 9. Lattice site update rate for NEC SX6+ and Cray X1 vector processors. A maximum domain size of 256x192x128 has been chosen owing to memory restrictions on one of the machines. For comparison a subset of the performance data of Intel Itanium 2 presented in Fig. 8 is included.

that the computationally intensive collision routine executes with almost the same performance independently of the domain size with up to about 75% of the peak performance. Already for the smallest domains, the length of the

loop in the *VectorProp* implementation is long enough ( $8^3 = 256$ ) to ensure efficient vector processing. However, for small domains, the ratio of surface area and total volume is large and therefore the routine for setting the boundary conditions at the borders consumes considerable time (up to 40% for the smallest test-case), which hurts overall performance. The effect of short loop length on vector performance becomes evident if performance numbers of *VectorProp* implementation and *PropOpt* version are compared. In the latter version the inner (vectorizable) loop length is  $Nx$  instead of  $Nx \times Ny \times Nz$  and the ( $Ny * Nz$  times) repeated start-up of the vector pipelines leads to a dramatic performance decrease below  $Nx = 64$  especially for the NEC system. Of course, if the vectorizable loops are sufficiently long ( $Nx = 256$ ) only minor performance variations between the two implementations are visible. For the Cray system we present results only for complete L2 cache bypass. Using the cache can improve performance for small domains at the cost of a performance degradation for larger ( $Nx > 32$ ) problem sizes [21].

It is worth mentioning that vector processors still outperform even the fastest COTS processor by a factor of more than 5 for reasonably large domains. If multiple processors are used within a shared-memory domain this gap grows significantly due to the bandwidth limitations of Intel/IBM based systems. Only for extremely small domains, where the on-chip cache of the Itanium processor can hold the working set, the COTS architecture is competitive. At this point the main difference between modern cache-based microprocessors and vector systems for memory intensive applications becomes evident: vector processors are essentially “cache-only” machines, i.e. their memory bandwidth is comparable with the typical cache bandwidths of standard CPUs. CFD codes like the LBM are memory-bound by default, so cache reuse is always limited and memory access dominates performance.

## 6 Conclusions

We have presented different implementations of a simple lattice Boltzmann kernel which is representative for many large scale LBM applications. Restricting ourselves to simple optimization techniques we found that the correct choice of the data layout is fundamental for achieving high performance. Even with full spatial blocking, the *collision optimized* layout was no match for a straightforward implementation of the *propagation optimized* layout, with the notable exception of the IBM Power4. Moreover, the latter layout also allows an efficient use of vector computers and cache-based microprocessors with identical code.

The Cray and NEC vector machines provide comparable performance levels and are a class of their own. Even the latest cache-based microprocessors cannot match performance of the vector processors. Their performance shortfall

by a factor 5–10 can be translated into a temporal backlog of at least 5 years concerning the sustainable single processor performance. The new Itanium 2 processor performs remarkably well and provides a significantly better single CPU performance than the new 64-bit enabled Intel Xeon or AMD Opteron processors. While the Xeon processor is substantially ahead of the Opteron in the single CPU race, we expect a significant decrease of the performance gap if both CPUs of standard two-way systems are used. IBM Power4 results fall usually between the IA64 and IA32 performance numbers but show an irregular characteristic which cannot be understood using basic considerations about cache structures.

Future work should extend our investigation to the impact of advanced blocking techniques — such as temporal blocking [16] — on the performance characteristics of the different data layouts as well as on the single processor/node performance.

Furthermore, as unstructured and / or adaptive grids are getting state-of-the-art for lattice Boltzmann approaches [22], optimization issues related to the then required vast indirect addressing become important and thus will also be addressed in future work.

## Acknowledgments

This work is financially supported by the Competence Network for Technical, Scientific High Performance Computing in Bavaria (KONWIHR). We thank F. Deserno, P. Lammers, T. Pohl, U. Rde, M. Wierse and R. Wolff for helpful discussion. Support from AMD, Intel and Cray is gratefully acknowledged.

## References

- [1] L. Oliker, J. C. A. Canning, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, R. V. d. Wijngaart, Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations, in: Proceedings of SC2003, CD-ROM, 2003.
- [2] D. Komatitsch, S. Tsuboi, C. Ji, J. Tromp, A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator, in: Proceedings of SC2003, CR-ROM, 2003.
- [3] Cray advanced technical workshop, Bologna (June 2004).
- [4] D. A. Wolf-Gladrow, Lattice-Gas Cellular Automata and Lattice Boltzmann Models, Vol. 1725 of Lecture Notes in Mathematics, Springer, Berlin, 2000.

- [5] S. Succi, *The Lattice Boltzmann Equation – For Fluid Dynamics and Beyond*, Clarendon Press, 2001.
- [6] S. Chen, G. D. Doolen, Lattice Boltzmann method for fluid flows, *Annu. Rev. Fluid Mech.* 30 (1998) 329–364.
- [7] Y. H. Qian, D. d’Humières, P. Lallemand, Lattice BGK models for Navier-Stokes equation, *Europhys. Lett.* 17 (6) (1992) 479–484.
- [8] D. Yu, R. Mei, L.-S. Luo, W. Shyy, Viscous flow computations with the method of lattice Boltzmann equation, *Progr. Aero. Sci.* 39 (2003) 329–367.
- [9] EXA Corp., Lattice boltzmann-based commercial flow solver “PowerFlow”, <http://www.exa.com/>.
- [10] T. Zeiser, M. Steven, H. Freund, P. Lammers, G. Brenner, F. Durst, J. Bernsdorf, Analysis of the flow field and pressure drop in fixed bed reactors with the help of lattice Boltzmann simulations, *Phil. Trans. R. Soc. Lond. A* 360 (1792) (2002) 507–520.
- [11] H. Chen, S. Kandasamy, S. Orszag, R. Shock, S. Succi, V. Yakhot, Extended Boltzmann kinetic equation for turbulent flows, *Science* 301 (5644) (2003) 633–636.
- [12] S. Chen, G. D. Doolen, K. G. Eggert, Lattice-Boltzmann fluid dynamics – a versatile tool for multiphase and other complicated flows, *Los Alamos Science* 22 (1994) 99–111.
- [13] T. Pohl, F. Deserno, N. Thürey, U. Rüde, P. Lammers, G. Wellein, T. Zeiser, Performance evaluation of parallel large-scale lattice-Boltzmann applications on three supercomputing architectures, in: *Proceedings of SC2004*, CD-ROM, 2004.
- [14] C. Körner, T. Pohl, U. Rüde, N. Thürey, T. Zeiser, Parallel lattice Boltzmann methods for CFD applications, in: *Numerical Solution of Partial Differential Equations on Parallel Computers*, Springer, accepted, 2005.
- [15] D. P. Ziegler, Boundary conditions for lattice Boltzmann simulations, *J. Stat. Phys.* 71 (5/6) (1993) 1171–1177.
- [16] T. Pohl, M. Kowarschik, J. Wilke, K. Igelberger, U. Rüde, Optimization and profiling of the cache performance of parallel lattice Boltzmann codes, *Par. Proc. Lett.* 13 (4) (2003) 549–560.
- [17] 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*, Springer, Berlin, 2001, pp. 115–122.
- [18] R. Argentini, A. Bakker, C. Lowe, Efficiently using memory in lattice Boltzmann simulations, *Future Generation Computer Systems* 20 (6) (2004) 973–980.

- [19] A. J. C. Ladd, Numerical simulations of particulate suspensions via a discrete Boltzmann equation. Part 1. Theoretical foundation, *J. Fluid Mech.* 271 (1994) 285–309.
- [20] C. Pan, J. F. Prins, C. T. Miller, A high-performance lattice Boltzmann implementation to model flow in porous media, *Comput. Phys. Commun.* 158 (2004) 89–105.
- [21] T. Zeiser, G. Wellein, G. Hager, S. Donath, F. Deserno, P. Lammers, M. Wierse, Optimization approaches and performance characteristics of lattice Boltzmann kernels on COTS and vector architectures, Tech. rep., Regionales Rechenzentrum Erlangen (2004).
- [22] J. Tölke, S. Freudiger, M. Krafczyk, An adaptive scheme for LBE multiphase flow simulations on hierarchical grids, submitted to *Computers & Fluids*, 2004.