

# Optimized Lattice Boltzmann Kernels as Testbeds for Processor Performance

Thomas Zeiser<sup>\*1</sup>, Gerhard Wellein<sup>1</sup>, Georg Hager<sup>1</sup>, Stefan Donath<sup>1</sup>, Frank Deserno<sup>2</sup>, Peter Lammers<sup>3</sup>, and Monika Wierse<sup>4</sup>

<sup>1</sup> Regionales Rechenzentrum Erlangen (RRZE), Martensstraße 1, D-91058 Erlangen, Germany

<sup>2</sup> Lehrstuhl für Systemsimulation (Informatik 10, LSS), Cauerstraße 6, D-91058 Erlangen, Germany

<sup>3</sup> Höchstleistungsrechenzentrum Stuttgart (HLRS), Allmandring 30, D-70550 Stuttgart, Germany

<sup>4</sup> Cray Computer GmbH, Germany

**Summary.** Delivering high sustained performance for scientific, memory-intensive applications is a well known problem in high performance computing (HPC). The main objective in the design of vector computers is to resolve this challenge. Commodity “off-the-shelf” (COTS) architectures do not mainly focus on HPC requirements, but dominate the HPC market due to their (often) moderate price-performance ratio. In our report we present a comprehensive survey of modern processor architectures ranging from IA32 compatible (Intel Xeon, AMD Opteron), superscalar RISC (IBM Power4), IA64 (Intel Itanium 2) to classical vector (NEC SX6) and novel vector (Cray X1) architectures. Using a kernel from the lattice Boltzmann method (LBM), we point out different architecture dependent optimization strategies and discuss single processor performance numbers. Our results demonstrate that vector systems can outperform COTS architectures by more than one order of magnitude. The NEC SX6 and Cray X1 achieve comparable performance levels on large problem sizes. Comparing different programming models of the LBM kernel shows the Cray X1 to deliver good performance even on the standard implementation of this kernel.

## 1 Introduction

Rapid advances in microprocessor technology have led to fundamental changes in the HPC market over the past decade. Commodity “off-the-shelf” (COTS) cache-based microprocessors arranged as systems of interconnected SMP nodes nowadays dominate the TOP500 list [1] due to their unmatched price/ peak performance ratio. However, it has also been acknowledged recently that the gap between sustained and peak performance for scientific applications on COTS platforms is growing continuously [2].

Although classical vector systems can bridge this performance gap especially for memory intensive codes, they only represent a tiny fraction (3.5 %) of all current TOP500 systems [1]. The combination of high development costs and a limited market volume for HPC systems should mainly account for this trend. Some authors also speculate that the ASCI program [3] has put very heavy emphasis on the use of COTS components [4]. Consequently, only one manufacturer of classical vector processors has survived which, however, set a landmark with the installation of the Earth Simulator (ES) using NEC SX6 vector technology. Achieving sustained performance numbers of several TFlop/s for a broad range of large scale applications [5–7], the ES has intensified discussions about the relevance of vector technology. A new class of vector computers has been introduced with the Cray X1. Although being very successful (10 TOP500 installations) in its first year of commercial availability, the Cray X1 must now demonstrate high sustained performance for a broad range of (vectorizable) applications.

However, not only the computers evolved. At the same time, new numerical methods have been developed and existing numerical algorithms and physical models have steadily been improved with respect to efficiency, applicability and validity. A recent method from the area of computational fluid dynamics (CFD) is the lattice Boltzmann method (LBM) [8]. Typical applications of the LBM include 3-D resolved flow in porous media [9, 10], turbulence research [11, 12] and multi-phase flows [13, 14]. These application areas are directly linked to systems with many degrees of freedom or equivalently large domain sizes. Therefore, they require both high floating point performance and also high bandwidth to main memory.

Owing to the high scientific potential of LBM for large scale applications, it is the aim of this paper to demonstrate architecture-dependent optimization strategies of a lattice Boltzmann kernel and to evaluate the single processor performance characteristics of COTS and vector processors. Parallelization strategies and parallel performance measurements are discussed in a complementary paper [15].

\* E-mail address: [thomas.zeiser@rrze.uni-erlangen.de](mailto:thomas.zeiser@rrze.uni-erlangen.de)

The remainder of this paper is organized as follows: In Sec. 2 we first present a comprehensive survey of the architectures 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 (Sec. 4) and discuss the measured single processor performance numbers together with theoretical estimates for the different machines (Sec. 5).

## 2 Architectural Specifications

In Tab. 1 we briefly sketch the most important single processor specifications of the architectures examined. COTS architectures are offered with a wide variety of different frequencies and cache sizes. The configurations as presented in the first group of Tab. 1 are those which are in common use in scientific computing centers. Concerning the memory architecture of COTS systems, we find a clear tendency towards on-chip caches which run at processor speed and provide high bandwidth as well as low latencies. The vector systems (second group of Tab. 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 numbers (Peak), maximum bandwidth of the memory interface (MemBW) and the sizes of the various cache levels are given. The L3 cache of the IBM Power4 processor and the L2 cache for the Cray X1 are off-chip caches, all other caches are on-chip. The L2 and L3 cache in the IBM p690 is shared by the two processors of a dual-CPU chip.

Platform	Single CPU specifications				
	Peak GFlop/s	MemBW GB/s	L1-cache kB	L2-cache MB	L3-cache MB
Intel Xeon DP (2.66 GHz)	5.3	4.3	8	0.5	–
Intel Itanium 2 (1.3 GHz)	5.2	6.4	16	0.25	3.0
IBM Power4 (1.7 GHz)	6.8	9.1	32	1.44	<i>32.0</i>
AMD Opteron (1.6 GHz)	3.2	5.3	64	1.0	–
NEC SX6 (500 MHz)	8.0	32.0	–	–	–
Cray X1 (800 MHz, 1 MSP)	12.8	34.1	–	<i>2.0</i>	–

### 2.1 Intel Xeon DP

The server variant (Xeon) of the Intel Pentium4 processor is widely used in COTS clusters and is well known for its high clock speed. The 32-bit Xeon processor can execute two double precision floating point (FP) operations (one multiply and one add) per cycle (SSE2). Using single precision data, the peak performance is doubled to 4 FP instruction per cycle. The on-chip caches of the Xeon DP (dual-processor variant) provide high bandwidth (32 Bytes per processor cycle) and low latencies (7 cycles) while data transfer from memory is limited to 4.3 GByte/s by the front-side bus frequency of 533 MHz. In standard dual-processor configurations, the CPUs have to share one bus, reducing the available memory bandwidth per processor by a factor of two.

The benchmark results reported were obtained with the latest Intel IA32 Fortran Compiler (Version 8.0) on a dual-processor node (using the Intel 7501 chipset) running Debian GNU/Linux 3.0.

### 2.2 Intel Itanium 2

The Intel Itanium/IA64 processor is a superscalar 64-bit CPU using the Explicitly Parallel Instruction Computing (EPIC) paradigm. In contrast to classical RISC systems, instructions are loaded in bundles of three. Only a limited number of combinations among memory, integer and floating point instructions per bundle are possible. The compiler is responsible for building the bundles and, moreover, has to specify groups of independent instructions which may be executed in parallel. Groups and bundles are two concepts that are, in a sense, orthogonal to each other, i.e. although the Itanium can issue two

bundles per cycle, a group can span any number of machine instructions. Of course this concept does not require any out-of-order execution support but demands high quality compilers to identify instruction level parallelism at compile time. While the first incarnation, the Itanium 1, has failed to become successful, the Itanium 2 is much more promising because of significant improvements in bandwidths, overall balance and improved compiler technology. The available clock frequencies range from 0.9 to 1.5 GHz and the on-chip L3 cache sizes from 1.5 to 6 MB. Two Multiply-Add units are fed by a large set of 128 FP registers, which is another important difference to standard microprocessors (typically 32 FP registers). Floating point data items bypass the L1 cache and are stored in the on-chip L2 and L3 caches, which provide high bandwidth (4 load or 2 load/2 store operations per cycle) and low latencies (5-6 cycles for L2; 10-12 cycles for L3). A large number of Itanium 2 systems from different vendors is available today. The basic building blocks of most systems used in scientific computing are dual-way nodes (SGI Altix, HP rx2600) or four-way nodes (NEC TX7, Bull NovaScale, HP rx5670) sharing one bus with 6.4 GByte/s memory bandwidth.

The system of choice in our report is a 28 processor SGI Altix3700 system (1.3 GHz; 3 MB L3 cache) at RRZE running RedHat Linux with SGI enhancements (“ProPack”). The lattice Boltzmann kernels were compiled with the Intel IA64 Fortran Compiler (Version 8.0).

### 2.3 IBM Power4

The IBM Power4 processor is a 64-bit superscalar (8-way fetch, 5-way sustained complete) out-of-order RISC processor with a maximum frequency of 1.7 GHz and two Multiply-Add units allowing for a peak performance of 6.8 GFlop/s. The basic difference to classical RISC systems is that two processors (cores) are placed on a single chip sharing high bandwidth (> 100 GByte/s) on-chip L2 cache, off-chip L3 cache and one path to memory. 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 L3 cache of 128 MB aggregated size. Although large in size, the L3 cache shows several drawbacks, e. g. long cache lines (512 Bytes), large latencies (up to 340 cycles [16]) and relatively low bandwidth (11.7 GByte/s for the 1.3 GHz CPU [17]). Moreover, the L3 cache line spans all four L3 caches of one MCM. If fully equipped, a 32-way IBM p690 node (1.3 GHz Power4) can offer an aggregate theoretical memory bandwidth of 110 GByte/s for read *and* 110 GByte/s for write operations.

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  $1/3$  of the core frequency.

### 2.4 AMD Opteron

The AMD Opteron processor is a 64-bit enabled version of the well-known AMD Athlon design. Maintaining full IA32 compatibility, the Opteron has architectural enhancements that provide a seamless transition to 64-bit software and at the same time improve overall system performance. These include:

- Integrated memory controller, eliminating the need for a separate north-bridge chip and reducing memory latency.
- Enlarged register set (compared to IA32) with eight additional 64-bit GP registers and eight additional 128-bit SSE registers.
- Support for Intel’s SSE2 instruction set.
- Three on-chip HyperTransport links (3.2 GByte/s each direction) for coupling to I/O and other Opteron processors.

The larger number of GP and FP registers reduces register pressure and enables more aggressive code optimization strategies than previously possible with IA32 designs. In SMP environments, Opteron processors have one path to memory per CPU due to the integrated memory controller. Consequently, the aggregated memory bandwidth scales with the CPU count. Cache-coherent shared-memory nodes with up to four processors can be easily built using the on-chip HyperTransport links.

Opteron processors are available with 64 kB of L1 and 1 MByte of L2 cache. The L1 data cache has two 64-bit ports for a peak bandwidth of 2 loads or stores per cycle. The unified L2 cache is designed as a so-called “victim cache”, receiving only cache lines that were evicted from L1. The core can sustainably execute one FP add and one FP multiply instruction per clock, allowing for a peak performance of 4 GFlop/s at the maximum clock frequency of 2 GHz. The maximum memory bandwidth per CPU is 5.3 GByte/s.

The benchmark results presented here have been measured at RRZE on a dual-Opteron workstation (1.6 GHz) with PC2100 memory modules providing only 80% of the memory interface. Another problem is posed by the fact that modern, standard-adhering and stable compilers are somewhat scarce for this CPU, especially for Fortran 90. The Intel *ifc* compilers produce IA32 executables which deliver a good performance on the Opteron, however, they are limited to 32-bit and do not make use of the additional registers the Opteron has. Therefore, we used the Portland Group (PGI) compiler which generates native 64-bit code for the Opteron, although the obtained performance currently is lower than with executables produced by the Intel compiler with optimization for a Xeon processor.

## 2.5 NEC SX6

From a programmers' view the NEC SX6 is a traditional vector processor with 8-way replicated vector pipes running at 500 MHz. One multiply and one add instruction per cycle can be executed by the arithmetic pipes delivering a peak performance of 8 GFlop/s. The memory bandwidth of 32 GByte/s allows for one load or store per Multiply-Add instruction. The processor contains 64 vector registers, each holding 256 64-bit words. For non-vectorizable instructions, the SX6 contains a 500 MHz scalar processor with a peak performance of 1.13 GFlop/s. Since the vector processor is significantly more powerful than the scalar unit, it is useless to run non-vectorized applications on a SX6 and thus vectorization is a must on this system. Each 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 Arctic Region Supercomputer Center (ARSC).

## 2.6 Cray X1

The Cray X1 was designed to be a fully decoupled, highly scalable system with up to 4096 powerful processors operating on a single global address space. The basic building block of the Cray X1 architecture is a *multi-streaming processor* (MSP) which one usually refers to as processor or CPU. The MSP itself comprises four processor chips, each incorporating a superscalar processor 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 (or eight single precision!) floating point operations and two memory operations. Running at a clock speed of 800 MHz, one MSP can thus perform up to 16 double precision floating point operations (12.8 GFlop/s) and issue 8 memory operations (51.2 GByte/s) per cycle. Note that the ratio of issued memory operations per issued Multiply-Add instruction is the same as for the NEC SX6 processor, but the memory interface of the MSP only delivers 34.1 GByte/s bandwidth and thus can not saturate the issued load instructions. The full bandwidth however is available for the L2 cache thus compensating this in situations where cache blocking can be applied.

At first glance long vectorized loops are, of course, 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 that each vector section takes a whole (much shorter) inner loop iteration of a nested loop, avoiding the rather long start-up times for wide-pipe vector processors. The L2 cache of the Cray X1 in addition provides a mechanism for efficiently executing on short vector loops as well; Sec. 5 will show this for small extents of the LBM domain.

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.

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

## 3 Basics of the Lattice Boltzmann Method

Within the last decade, the lattice Boltzmann method (LBM) [8, 18–20] has evolved into a promising alternative for the numerical simulation of (time-dependent) incompressible flows. Whereas conventional CFD methods are based on a discretization of macroscopic differential equations (in particular the Navier-Stokes equations), the LBM follows a *bottom-up* approach by calculating the evolution of a particle distribution function. For a recent review of the method including a detailed comparison with conventional Navier-Stokes solvers, the reader is referred to [20].



A trivial implementation would consist of three nested loops over the three spatial dimensions and treat the collision step independently from the propagation process. That means, it would read the values of the current time step from the local cell, execute the relaxation and write the results back to a temporary array as this can be done independently for all cells. In a separate nested loop which only would contain memory operations, these values then would be loaded again and be propagated back to adjacent cells in the original array.

Quite a number of improvements are possible starting from this trivial implementation. Some of them are discussed in more detail in the following subsections.

#### 4.1 Standard Version

The collision process can be combined with the propagation step. To keep the implementation simple and not to care about the order of updating neighboring cells in the propagation step, two copies of the  $f_i$  array are again kept in memory — one for the values before propagation and the other for the values after propagation. 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 time-step the two arrays are switched, i.e. the source becomes the destination and vice-versa. That means, depending on the implementation, the propagation step is realized as first or last step of the iteration loop, resulting in a “draw” or “push” scheme of the update process:

##### Draw:

- read distribution functions  $f_i$  from *adjacent* cells
- calculate  $\rho$ ,  $\mathbf{u}$  and  $f_i^{\text{eq}}$
- write updated  $f_i^*$  values to *current* cell

##### Push:

- read distribution functions  $f_i$  from *current* cell
- calculate  $\rho$ ,  $\mathbf{u}$  and  $f_i^{\text{eq}}$
- write updated  $f_i^*$  values to *adjacent* cells

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.

The data for the values of the distribution functions are preferentially stored in one array of dimension 5: three spatial coordinates, the discrete-velocity direction  $i$  and the time-step ( $t$  or  $t^*$ ). The order of the indices can have substantial performance impacts. For the remainder of the paper, the algorithm with the propagation collapsed into the collision step and pushing data with three nested loops over the spatial directions is called *standard* implementation (Fig. 2). Only the index orders  $(x, y, z, i, t)$  (*flipped*) and  $(i, x, y, z, t)$  (*non-flipped*) have been considered with the first index moving fastest as the benchmark kernel was implemented in Fortran.

#### 4.2 Vector Optimization

For vector architectures, long inner loops are preferential. In the case of the LBM, the three nested spatial loops  $(x, y, z)$  can be fused into just one large loop ( $m$ ) which is fully vectorized (Fig. 3). However, in order to be able to do the propagation even at the boundaries of the computational domain, an additional ghost-layer is added around the actual cubic calculation box together with an appropriate mask which blocks out the ghost-cells from the calculation and propagation process.

```

real(kind=8), dimension(0:xE+1,0:yE+1,0:zE+1,0:18,0:1) :: f
logical, dimension(1:xE,1:yE,1:zE) :: fluidCell
real(kind=8) :: dens, ne, ...
do z=1,zE; do y=1,yE; do x=1,xE
  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)*ImOmega+ne0
    f(x+1,y+1,z , 1,tN)=f(x,y,z, 1,t)*ImOmega+ne1
    ...
    f(x ,y-1,z-1,18,tN)=f(x,y,z,18,t)*ImOmega+ne18
  endif
enddo; enddo; enddo

```

Fig. 2. Layout of the “standard version”.

```

real(kind=8), dimension(0:(1+xE)*(1+yE)*(1+zE),0:18,0:1) :: f
logical, dimension(0:(xE+1),0:(yE+1),0:(zE+1)) :: fluidCell
real(kind=8) :: dens, ne, ...
do m=0, (1+xE)*(1+yE)*(1+zE)
  if ( fluidCell(m) ) then
    ! read distributions from local cell and
    ! calculate moments
    dens=f(m,0,t)+f(m,1,t)+f(m,2,t)+...
    ...
    ! compute non-equilibrium parts based on local moments
    ne0=...
    ...
    ! write updates to neighboring cells
    f(m , 0,tN)=f(m, 0,t)*ImOmega+ne0
    f(m+1+(xE+1) , 1,tN)=f(m, 1,t)*ImOmega+ne1
    ...
    f(m -(xE+1)-(xE+1)*(yE+1),18,tN)=f(m,18,t)*ImOmega+ne18
  endif
enddo

```

Fig. 3. Layout of the “vector version”.

### 4.3 Cray X1 MSP

As pointed out in Sec. 2.6, controlling the L2-cache for certain variables influences the performance on the Cray X1. To make use of this in the LBM loop, it has to be taken care that the values  $f_i$  at the two different time levels are passed in two different arrays, such that they can be treated separately by compiler directives. The best performance on the Cray X1 is achieved with the flipped vector version of the collision routine where the “new” distribution values are not allocated in cache while writing back to memory. However, since these values are reused in the next time-step, this cache treatment in general hurts the performance for small problem sizes, especially for the non-flipped version, as demonstrated in Sec. 5.

### 4.4 Cache Optimization

On cache-based machines, the path to the main memory is a serious bottleneck for memory intensive applications like LBM codes. Therefore, a major aim of an efficient implementation is the increase of cache reuse — if necessary even putting up with additional operations.

A data layout with the 19 discrete-velocity directions  $i$  as first index (non-flipped) results in loading few cache lines and continuous access to the data of a cell. However, the results have to be stored to non-continuous memory areas and therefore involve many different cache lines. Additionally, the index  $x$  of the inner loop is not the one which changes fastest.

Using  $(x, y, z, i, t)$  as data layout (flipped), makes the  $x$  index moving fastest. In contiguous memory areas, complete  $x$  lines for the different discrete-velocity directions  $i$  can be found. To exploit this fact, the computational work within the  $x$  loop is divided into several parts. First of all, current distribution values of a complete  $x$  line are read and the different moments which are required later on for the calculation of the equilibrium distribution and the non-equilibrium part are precomputed and stored in temporary arrays. Then in an other loop, the non-equilibrium parts are calculated. Finally, in separate loops for all discrete velocity directions  $i$ , the relaxation is executed and the results are written back to the adjacent cells for the next time-step. By line-wise writing back the data, contiguous memory areas are accessed. The basic principles of this implementation are summarized in Fig. 4.

Starting at a certain architecture-dependent  $x$  size, blocking of the inner loop ( $x$ ) can be advantageous in order to ensure that all temporary data remains in the cache. The additional modifications of the code are outlined in Fig. 5. Of course, the blocking can be easily extended to three dimensional 3-way blocking.

```

real(kind=8), dimension(1:xE,1:yE,1:zE,0:18,0:1) :: f
real(kind=8), dimension(1:xE) :: dens, ...
real(kind=8), dimension(1:xE,0:18) :: ne
do z=1,zE; do y=1,yE
! read distributions from local cell and calculate moments
do x=1,xE
dens(x)=f(x,y,z,0,t)+f(x,y,z,1,t)+f(x,y,z,2,t)+...
...
enddo
! compute non-equilibrium parts based on local moments
do x=1,xE
ne(x,0) = ...
...
enddo
! write updates to neighboring cells;
! separate loops for all directions
do x=1,xE
if ( fluidCell(x,y,z) ) then
f(x ,y ,z , 0,tN)=f(x,y,z, 0,t)*ImOmega+ne(x, 0)
endif
enddo
...
do x=1,xE
if ( fluidCell(x,y,z) ) then
f(x ,y-1,z-1,18,tN)=f(x,y,z,18,t)*ImOmega+ne(x,18)
endif
enddo
enddo; enddo
    
```

Fig. 4. Cache-optimized code; risc version.

```

do xx=1,xE, BLOCKSIZE
do z=1,zE; do y=1,yE
! read distributions from local cell ...
do x=xx,min(xE,xx+BLOCKSIZE-1)
dens(x)=f(x,y,z,0,t)+f(x,y,z,1,t)+...
...
enddo
! compute non-equilibrium parts ...
do x=xx,min(xE,xx+BLOCKSIZE-1)
ne(x,0)=...
...
enddo
! write updates to neighboring cells;
! separate loops for all directions
do x=xx,min(xE,xx+BLOCKSIZE-1)
if ( fluidCell(x,y,z) ) then
f(x ,y ,z , 0,tN)=f(x,y,z, 0,t)*ImOmega+ne(x, 0)
endif
enddo
...
do x=xx,min(xE,xx+BLOCKSIZE-1)
if ( fluidCell(x,y,z) ) then
f(x ,y-1,z-1,18,tN)=f(x,y,z,18,t)*ImOmega+ne(x,18)
endif
enddo
enddo; enddo
enddo
    
```

Fig. 5. Cache-optimized version with blocking.

## 4.5 Other Optimization Strategies

In literature, also some other recent optimization strategies can be found which aim at reducing the memory consumption and/or improving performance.

Pohl *et al.* [23] 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 certain architectures.

Pohl *et al.* [23] and Schulz *et al.* [24] presented two different “compressed grid” approaches, i.e. through a careful consideration of the order of executing the propagation step of the individual cells, they manage to get almost completely rid of the second  $f_i$  array. This strategy reduces the total memory consumption almost by a factor of two. However, it is not clear yet whether the cache reuse is really improved as the propagation step of [24] has to be done direction dependent.

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.

Argentini *et al.* [25] use the non-BGK model of Ladd [26] and thus succeed in storing only 9 moments of the distribution functions instead of all the distribution values itself. However, they admit that an application of this idea to the common BGK model is not (transparently) possible.

Pan *et al.* [27] as well as Schulz *et al.* [24] 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” and use some lists with the data of the distribution functions and the connectivity. Thus, only memory for the data of fluid cells consisting of the distribution functions and pointers to all  $N$  neighbors are required. For low porosities, a lot of memory can be saved in this way, however, the memory access patterns get much more complicated and include vast indirect addressing which probably reduces the performance on vector machines as well as the cache reuse on RISC machines. Details of ordering the data in the lists (e.g. Morton ordering as shown in [27]) can lower the performance impact.

## 5 Results

In the following, an estimation of the theoretically achievable performance on the different platforms as well as measurements of the performance with the different implementation and optimization strategies outlined above are presented and 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 only.

### 5.1 Estimation of Achievable Performance

Based on characteristic quantities of the benchmarked architectures (Table 1), an estimation of possible performance numbers can be made. Considering the theoretical memory bandwidth as the limiting factor, the number of bytes per cell  $B$  to be loaded or stored during an update has to be determined. Assuming a D3Q19 model and *write allocate*-behavior when storing to memory,  $B = 3 \cdot 19 \cdot 8$  bytes = 456 bytes per cell and time step must be transferred by the memory bus system. Without the *write allocate* requirement (e.g. on NEC SX6), only 304 bytes have to be transferred. The attainable performance  $P$  in MLUPS therefore accounts to

$$P = \frac{\text{MemBW}}{B}. \quad (4)$$

In contrast, assuming the peak performance of the processor to be the performance bottleneck, the performance  $P$  yields to

$$P = \frac{\text{Peak Perf.}}{F}, \quad (5)$$

where  $F$  is the number of floating point operations per cell.

While the memory bandwidth is given by the architecture, the average number of Flops per cell is slightly architecture and compiler dependent. Counting the operations in the source code as well as looking at profiling output using the highest available optimization level on different architectures, gives an average of 200 Flops per cell update. That means 1 GFlop/s is equivalent to about 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  $100^3$  or  $128^3$  for all benchmarked architectures.

**Table 2.** Comparison of the maximal performance owing to the peak performance (column Peak) or memory bandwidth (column MemBW) and the maximum of the measured update rates for the D3Q19 model and a domain size of  $100^3$  or  $128^3$ .

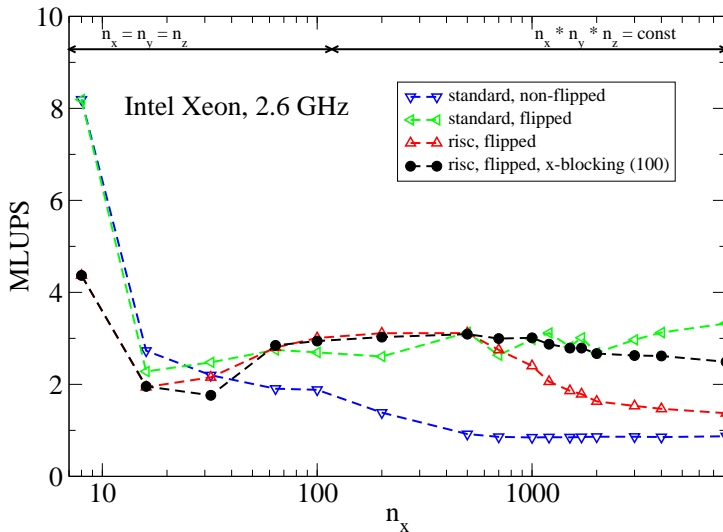
Platform			max. MLUPS		measurement	
	GFlop/s	GByte/s	Peak	MemBW	max. MLUPS achieved for $100^3$ / $128^3$	
Intel Xeon (2.6 GHz)	5.3	4.3	26.5	9.4	3.0	risc, flipped, $100^3$
Intel Itanium 2 (1.3 GHz)	5.2	6.4	26.0	14.0	6.9	risc, flipped, $100^3$
IBM Power4 (1.7 GHz)	6.8	9.1	34.0	20.0	4.2	vector, non-flipped, $128^3$
AMD Opteron (1.6 GHz)	3.2	5.4	16.0	11.8	2.4	risc, flipped, blocked, $100^3$
NEC SX6 (500 MHz)	8.0	32.0	40.0	105	35.4	vector, flipped, $128^3$
Cray X1 (1 MSP)	12.8	34.1	64.0	112	34.9	vector, flipped, no-cache, $128^3$

## 5.2 Measurements

For clearness reasons, from all the measurements done, only those algorithms are plotted in the following diagrams which give the best performance or show a remarkable behavior on the respective machine.

### Workstations with Intel Xeon and AMD Opteron CPU

The Intel Xeon CPUs are well known for their high clock frequencies. Therefore, for very small system sizes, a quite high update rate can be obtained with all *standard* versions. With increasing system size or length of the inner loop, the performance of the *non-flipped standard* version is decreasing quickly. The *flipped standard* version remains at rather high level over the complete range investigated. The *cache optimized RISC* algorithm slightly outperforms the *standard* version in the range of medium loop lengths. *Blocking* of the inner loop ensures that the performance does not decrease for large inner loops.

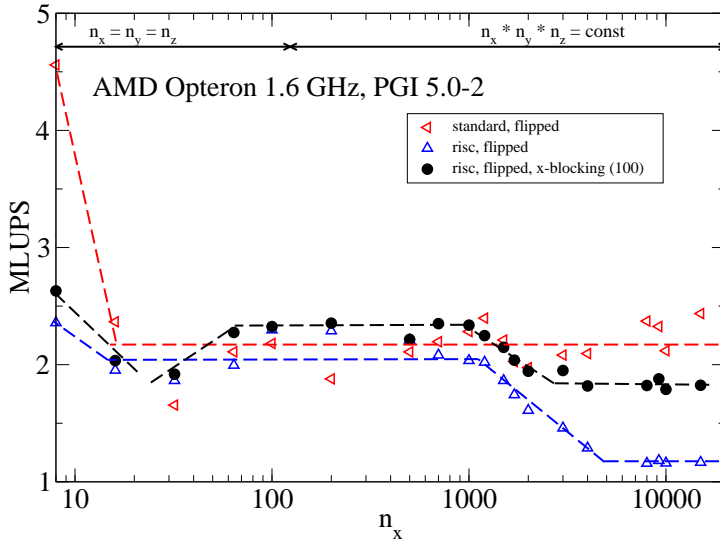


**Fig. 6.** Lattice site update rate on an Intel Xeon (with Intel Fortran compiler 8.0) as function of the length  $n_x$  of the inner loop.

For a system size up to  $n_x = 128$  all three spatial directions are equally increased. For larger values of  $n_x$ , the dimensions of the other two spatial directions have been reduced to give a constant total system size still fitting into the RAM of the machine.

The measurements on the AMD Opteron machine scattered heavily although no other jobs or users were active. The lines drawn in Fig. 7 try to show the general trend. Good performance is achieved with the *flipped standard* algorithm for the complete range of system sizes investigated. The *cache optimized RISC* version does not improve things, however, when used together with *blocking* of the inner loop, slightly higher update rates than with the *standard* version are obtained in the intermediate range.

Although the clock frequency of the Opteron system is much lower and the PGI compiler currently is less advanced, the Opteron reaches a similar overall performance as the Intel Xeon does. Using the Intel *ifc* compiler (with Xeon optimizations switched on), a slightly better performance could be obtained, however, the *ifc* compiler is limited to 32-bit and does not make use of the additional registers the Opteron has. It is therefore expected that the Opteron and its compilers still have quite some margin for further performance improvements — although first tests with the PGI compiler 5.1 did not show much difference yet.



**Fig. 7.** Lattice site update rate on an AMD Opteron system (with Portland Group compiler 5.0-2 in 64-bit mode) as function of the length  $n_x$  of the inner loop.

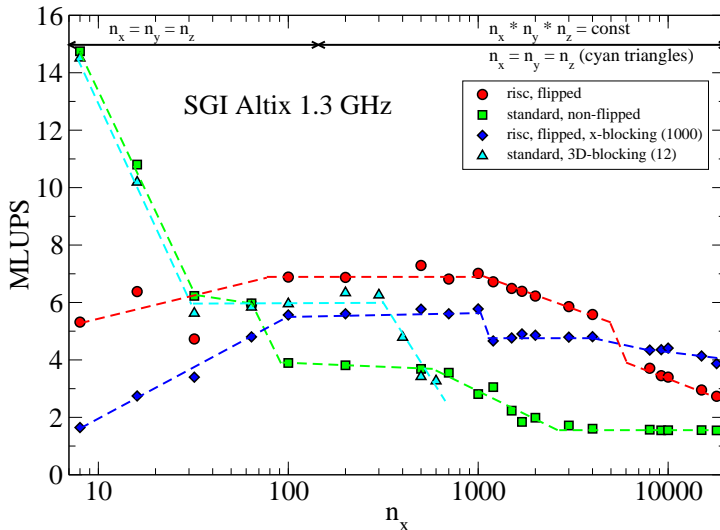
For a system size up to  $n_x = 128$  all three spatial directions are equally increased. For larger values of  $n_x$ , the dimensions of the other two spatial directions have been reduced to give a constant total system size still fitting into the RAM of the machine.

### SGI Altix with Intel Itanium 2 CPU

For small domain sizes, *all standard* versions show a very high update rate owing to the efficient cache usage. *Flipping* does not have a noticeable effect, also *blocking* does not significantly alter the results (as expected). The strong change of the slope at a system size of  $32^3$  (about 10 MB data) marks the point where the data no longer fits into the caches. The *risc-flipped* version gives a poor update rate for these small domains. When switching on *blocking*, the update rate is even worse in this case. This finding is not surprising as the data more or less fits into the caches and therefore the *risc* optimization with the additional loops and temporary arrays only cause additional overhead.

Already at small/medium system sizes, the *standard* version drops to a relatively low level which further decreases for large values of  $n_x$ . However, for a wide range of medium/large loop lengths (while keeping the total system size constant beyond  $n_x = 128$  by shrinking the other dimensions) the *non-blocked, flipped risc* version is superior. *x-blocking* only improves the results slightly for very large values of  $n_x$ .

Going to really huge domains by keeping cubic domains, i.e. always increasing all directions by the same factor, gives a slightly different behavior and *3-D blocking* of the *non-flipped standard* algorithm with small blocking factors pays out. Through the 3-way blocking, the update rate thus can be kept at a high level even for huge domain sizes. A *3-way blocking* of the *risc-flipped* version, however, is not efficient at all. The significant decrease of the cyan curve (triangles) in Fig. 8 at a system size of  $300^3$  is related to the fact that additionally non-local memory of the SMP system has to be used which decreases the available memory bandwidth.

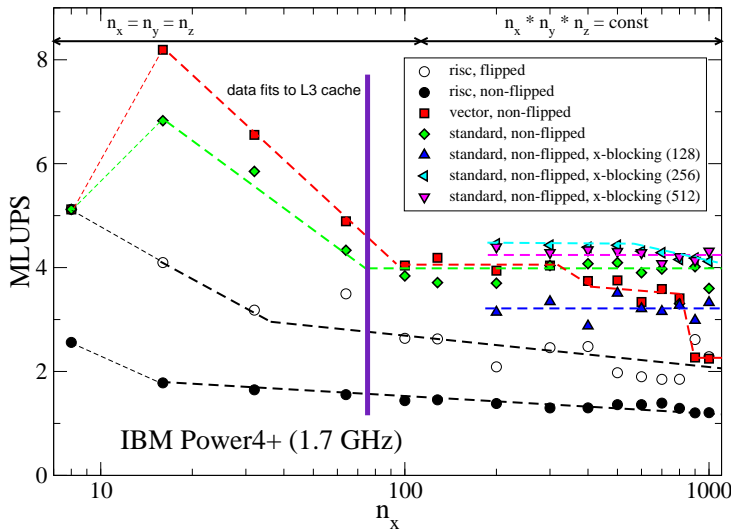


**Fig. 8.** Lattice site update rate on the SGI Altix with 16 GB per CPU brick using a single Intel Itanium 2 CPU (1.3 GHz, 3 MB L3 cache).

For the 3-D blocking algorithm, the system size has been increased equally in all three directions for all system sizes. For all other implementations shown in this diagram, for values of  $n_x$  larger than 128, the dimensions of the other two spatial directions have been reduced to give comparable results with the smaller machines used in our survey.

### IBM p690 with IBM Power4+ CPU

The results obtained on one CPU of a IBM Power4+ p690 node are disappointing. The *cache optimized risc* version did not show any improvement. The *non-flipped* data layout proved to be the best for this architecture. For small and intermediate system sizes, the *vector* variant is the best one, however, for very large lengths of the inner loop (while keeping the total domain size constant), the *standard* version is getting better. A remarkable point is that — in contrast to the Itanium or IA32 systems — the IBM does not obtain the highest performance for the smallest system size. Obviously, the IBM has a quite large overhead for starting loops or calling subroutines and benefits from the large cache sizes. The change of slope between a domain size of  $64^3$  and  $100^3$  which can be clearly detected for the *non-flipped vector* and *standard* version can be attributed to the aggregated L3 cache size of a p690 MCM. Tests with different *blocking* sizes of the inner loop showed that an intermediate blocking factor can give a certain improvement for large domain lengths. However, although the theoretical performance estimate for a Power4+ CPU results in much higher update rates than an Itanium 2, practice shows a completely different behavior making the IBM p690 not the ideal choice for CFD applications.



**Fig. 9.** Lattice site update rate of a single IBM Power4+ CPU on a fully equipped IBM p690 node.

For a system size up to  $n_x = 128$  all three spatial directions are equally increased. For larger values of  $n_x$ , the dimensions of the other two spatial directions have been reduced to give comparable results with the smaller machines used in our survey. The vertical bar marks the system size which should fit into the aggregated L3 cache of a MCM of the p690 node.

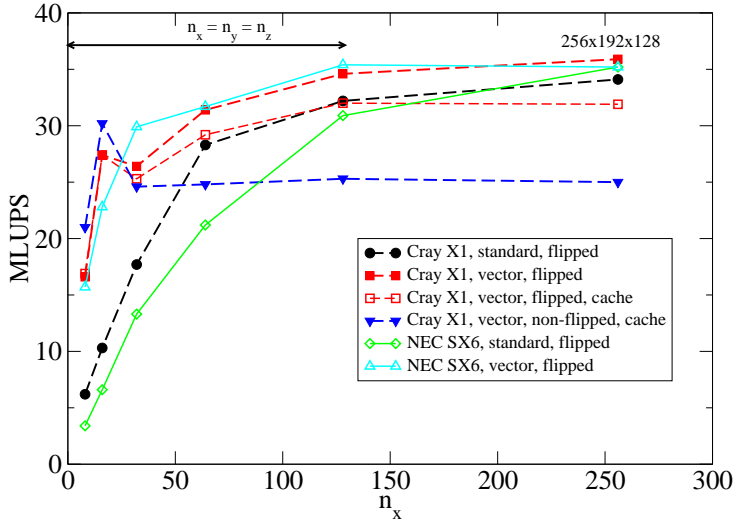
### Vector systems: NEC SX6 and Cray X1

A clearly remarkable observation for the vector systems is the fact that there is no performance decrease for very large domain sizes (tested up to  $512^3$ ). It is even the other way round. With increasing system size, the performance is increasing. At first sight, one might attribute this fact to the overhead of loading short vectors when dealing with small domains. However, a detailed profiling of the code on both vector systems showed that the computational intensive collision routines executes with almost the same performance independently of the domain size with up to about 75% of the peak performance. Already for the smallest domain sizes, the vector length of the *vector-optimized* algorithm is long enough to ensure an efficient vector processing. However, for small domain sizes, the ratio between surface area and total volume is large and therefore, the routine for setting the boundary conditions at the borders of the domain consume considerable time (up to 40% for the smallest test-case) which decreases the shown total performance of the benchmark application.

It is remarkable that the index order has a significant effect even on the NEC SX6. It turned out that the *flipped* version shown in Fig. 10 is better than the *non-flipped* data layout. The reason is not clear in the moment as there were no significant bank conflicts or I/O-Cache misses in either case.

On the Cray X1, there are even bigger differences between the various algorithms and data layouts. For all measurements the distribution values after collision (i.e.  $f_i^*$  at time  $tN$ ) are not allocated in the cache by default. If this array is allocated in cache, we denote this with 'cache'. The *flipped vector* algorithm with the special *no allocating cache* treatment for the array  $f_i^*$  performs best for large problem sizes. This special cache treatment is helping for domains with  $32 \times 32 \times 32$  or more cells. For very small problem sizes the *non-flipped vector* variant with  $f_i^*$  allocated in the L2-cache can outperform the *vector* version, since there is a lot of cache reuse of the variable  $f_i^*$  possible.

It is remarkable that the standard version of the algorithm without any modifications already performs rather well on the Cray X1. As the machine is quite new, it is expected that the obtained performance will further increase in the future with updated releases of the compilers and perhaps other adaptations of the algorithms to specific features of the machine.



**Fig. 10.** Lattice site update rate on a single NEC SX6 or Cray X1 vector CPU.

The system size has been increased equally in all three directions for all system sizes (except the last which corresponds to 256x192x128 owing to memory restrictions on one of the benchmark machines). The poor performance for small system sizes is due to the overhead of boundary handling — not due to limited vector performance in the computational intensive collision routine.

## 6 Conclusions

For the LBM, we demonstrated that the “tailored” vector architectures, NEC SX6 and Cray X1, provide comparable single processor performance which cannot easily be matched by other systems which are behind by a factor of 5–10. The new Itanium 2 processor performs remarkably well and provides a significant performance gain compared to other microprocessors such as the IBM Power4, Intel Xeon or AMD Opteron. The “computer pyramid” built by COTS clusters at the base, tailored vector HPC systems at the top and clusters of shared memory systems with tailored components (e.g. high-speed interconnects) for the gap in between, seems to be the only suitable model to meet the different requirements of HPC users and codes. However, for large scale CFD applications, only tailored high-end HPC vector systems seem to be able to provide a sustained performance of more than 1 TFlop/s with reasonable processor counts (e.g. 256 vector processors [28]). Comparable performance would require — even assuming ideal parallel speedup — several thousands of cache-based microprocessors connected via expensive high-speed interconnects [15, 28].

## Acknowledgments

This work is financially supported by the Competence Network for Technical, Scientific High Performance Computing in Bavaria (KONWIHR).

## References

1. Top 500 list, available at <http://www.top500.org> (November 2003).
2. 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.
3. The asci program, <http://www.llnl.gov/asci/>.
4. A. J. van der Steen, J. Dongarra, <http://www.phys.uu.nl/~steen/web03/overview.html>.
5. H. Sakagami, H. Murai, Y. Seo, M. Yokokawa, 14.9 TFlops three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator, in: Proceedings of SC2002, CD-ROM, 2002.
6. S. Shingu, A 26.58 TFlops global atmospheric simulation with the spectral transform method on the Earth Simulator, in: Proceedings of SC2002, CD-ROM, 2002.
7. 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.
8. S. Chen, G. D. Doolen, Lattice Boltzmann method for fluid flows, *Annu. Rev. Fluid Mech.* 30 (1998) 329–364.

9. R. J. Hill, D. L. Koch, Moderate-Reynolds-number flow in a wall bounded porous medium, *J. Fluid Mech.* 453 (2002) 315–344.
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. J. G. M. Eggels, Direct and large-eddy simulation of turbulent fluid flow using the lattice-Boltzmann scheme, *Int. J. Heat and Fluid Flow* 17 (1996) 307–323.
13. 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.
14. X. He, S. Chen, R. Zhang, A lattice Boltzmann scheme for incompressible multiphase flow and its application in simulation of Rayleigh-Taylor instability, *J. Comput. Phys.* 152 (2) (1999) 642–663.
15. T. Pohl, N. Thürey, F. Deserno, P. Lammers, U. Rüde, Parallel performance of large-scale lattice-Boltzmann applications, submitted to SC2004.
16. S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O’Connell, W. Weir, The Power4 Processor Introduction and Tuning Guide, <http://www.ibm.com/redbooks/>, IBM, 2001.
17. F. Krämer, private communication, IBM (2003).
18. D. A. Wolf-Gladrow, Lattice-Gas Cellular Automata and Lattice Boltzmann Models, Vol. 1725 of Lecture Notes in Mathematics, Springer, Berlin, 2000.
19. S. Succi, *The Lattice Boltzmann Equation – For Fluid Dynamics and Beyond*, Clarendon Press, 2001.
20. 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.
21. Y. H. Qian, D. d’Humières, P. Lallemand, Lattice BGK models for Navier-Stokes equation, *Europhys. Lett.* 17 (6) (1992) 479–484.
22. D. P. Ziegler, Boundary conditions for lattice Boltzmann simulations, *J. Stat. Phys.* 71 (5/6) (1993) 1171–1177.
23. 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.
24. 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.
25. R. Argentini, A. Bakker, C. Lowe, Efficiently using memory in lattice Boltzmann simulations, *Future Generation Computer Systems* (2004) in press.
26. 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.
27. 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.
28. F. Deserno, G. Hager, F. Brechtfeld, G. Wellein, Performance of scientific applications on modern supercomputers, in: S. Wagner, W. Hanke, A. Bode, F. Durst (Eds.), *High Performance Computing in Science and Engineering*, Munich 2004, Springer, in press.