Performance comparison of different parallel lattice Boltzmann implementations on multi-core multi-socket systems

S. Donath* and K. Iglberger

System Simulation – Computer Science 10 (LSS),
University of Erlangen-Nuremberg, Germany
E-mail: stefan.donath@informatik.uni-erlangen.de
E-mail: klaus.iglberger@informatik.uni-erlangen.de
*Corresponding author

G. Wellein, T. Zeiser and A. Nitsure

Regional Computing Center of Erlangen (RRZE),
University of Erlangen-Nuremberg, Germany
E-mail: gerhard.wellein@rrze.uni-erlangen.de
E-mail: thomas.zeiser@rrze.uni-erlangen.de

U. Rüde

System Simulation – Computer Science 10 (LSS),
University of Erlangen-Nuremberg, Germany
E-mail: ulrich.ruede@informatik.uni-erlangen.de

Abstract: In this report, we discuss the performance behaviour of different parallel lattice Boltzmann implementations. In previous works, we already proposed a fast serial implementation and a cache oblivious spatial and temporal blocking algorithm for the lattice Boltzmann method (LBM) in three spatial dimensions. The cache oblivious update scheme has originally been proposed by Frigo et al. The main idea is to provide maximum performance results for stencil-based methods by dividing the space–time domain in an optimal way, independently of any external parameters, such as cache size. In view of the increasing gap between processor speed and memory performance, this approach offers a promising path to increase cache utilisation. We present results for the shared memory parallelisation of the cache oblivious implementation based on task queueing in comparison to the iterative standard implementation, thereby focusing on the special issues for multi-core and multi-socket systems.

Keywords: lattice Boltzmann; cache optimisation; cache oblivious; multi-core; task queueing; shared-memory parallelisation.

Reference to this paper should be made as follows: Donath, S., Iglberger, K., Wellein, G., Zeiser, T., Nitsure, A. and Rüde, U. (2008) 'Performance comparison of different parallel lattice Boltzmann implementations on multi-core multi-socket systems', *Int. J. Computational Science and Engineering*, Vol. 4, No. 1, pp.3–11.

Biographical notes: Stefan Donath is a research assistant at LSS working on his PhD about free-surface extensions to the lattice Boltzmann method.

Klaus Iglberger works as a research assistant on a PhD in rigid body dynamics with coupling to fluid flow. Both PhD theses are part of the waLBerla lattice Boltzmann project.

Gerhard Wellein received his PhD in physics from the University of Bayreuth and is now head of the high-performance computing group at RRZE.

Thomas Zeiser has finished his PhD in Chemical Engineering at the University of Erlangen and works on code performance optimisation at RRZE.

Aditya Nitsure completed his Master's thesis at LSS and worked as an Assistant at RWTH Aachen.

Ulrich Rüde is Professor in the Department of Computer Science and Head of the LSS. His fields of interest include high performance computing, computational science and engineering, and adaptive multilevel algorithms.

Copyright © 2008 Inderscience Enterprises Ltd.

1 Introduction

In the last decade, the advances in microprocessor technology have continuously increased the clock frequencies and single-processor peak performance of modern CPUs as well as the capacity of main memory. However, the bandwidth between processor and main memory is growing only very slowly and falls behind the actually required performance. Even sophisticated levels of caches close to the CPU cannot bridge this gap for memory intensive applications. With the current transition from single-core to dual-/multi-core CPUs, the memory gap will become even more pronounced despite recent changes in the architectures of memory subsystems.

To decrease the performance penalty and to optimally utilise the small but fast caches, spatial and temporal locality of data and algorithms must be exploited to allow reuse of cached data and thus reduce the amount of data (i.e., complete cache lines) transferred between CPU and main memory. A basic requirement for this is the choice of reasonable data layouts, which allow fast and continuous access to large blocks of data. Based on these, traditional cache blocking techniques apply 2-/3-way spatial blocking to improve the spatial locality or 4-way space–time blocking to additionally improve temporal locality of the data (Rivera and Tseng, 2000; Kowarschik, 2004). However, the choice of good blocking factors depends on the size and speed of the different cache levels. In contrast to these explicit blocking schemes, cache oblivious techniques as proposed by Frigo et al. (1999) try to remove this dependency on cache size and thus are promising candidates for high performance algorithms. However, as will be shown in Section 6, the implicit blocking of data and automatically adapted locality introduces new problems for multi-core and multi-socket systems.

Computational Fluid Dynamics (CFD), in general, and lattice Boltzmann methods (LBM) (Succi, 2001) for Direct Numerical Simulations (DNS), in particular, are data and memory bandwidth demanding applications. The LBM is known for its computational efficiency and easy extensibility and is therefore predestined for a performance study on modern and future computer architectures. Starting from a naïve implementation that separates the two basic steps of the LBM, i.e., *collision* and *propagation* step, and generates additional pressure on the memory subsystem by touching all data twice (cf. the discussion in Wellein et al. (2006b)), many different optimisation approaches can be found in literature. One possible optimisation is discussed in Stockman and Glass (1998) and Schulz et al. (2002): using suitable data layouts, the separated propagation may be done for many cells (i.e., all distribution functions of a certain direction) by simply changing a pointer and, thus, reducing the costs of propagation significantly. Another approach is the use of a *two-grid implementation (toggle arrays)*, which intuitively allows to combine the collision and propagation into one single step. Depending on the architecture, a *collide–propagate (push)* or *propagate–collide (pull)* sequence might be slightly better (Iglberger, 2003). *Compressed grid methods* (Pohl et al., 2003) allow to reduce the total memory footprint by almost 50% even with combined collision and propagation.

As shown in a previous paper (Wellein et al., 2006b), the data layout, i.e., the position of the *direction index* within the five-dimensional array tuple (three spatial coordinates, one direction index, one time index), has a tremendous influence on the performance of LBM in three spatial dimensions. The *structure-of-arrays* data layout (i.e., the values of one direction are consecutive in memory for all nodes) which is optimised for the propagation of distribution functions easily exceeds the performance of the *array-of-structures* data layout (i.e., the values of all distribution functions of one cell are consecutive in memory) which is optimised for the collision of distribution functions, even if spatial blocking is applied in the latter case (Wellein et al., 2006b).

Based on the results of previous papers, where we already demonstrated that the cache oblivious Frigo approach can successfully be applied to a LBM code with three spatial dimensions (Nitsure et al., 2006) and that even for the recursive structure of the algorithm a shared memory parallelisation is possible using a task queue model (Zeiser et al., 2008), the aim of the present paper is to investigate the impact of the cache oblivious scheme on multi-core and multi-socket systems.

The remainder of this paper is organised as follows: In Section 2 we first give a brief introduction to the lattice Boltzmann method. The fundamentals of the sequential cache oblivious lattice Boltzmann implementation are described in Section 3 while details on the parallel implementation can be found in Section 4. In Section 5 we summarise architectural characteristics of the systems used in our study. The results section (Section 6) finally discusses the issues of a simple iterative and the cache oblivious LBM implementation on multi-core multi-socket systems and gives hints to achieve optimal performance on several systems.

2 The lattice Boltzmann method

The LBM (Succi, 2001) is a stencil based approach from computational fluid dynamics for solving (time-dependent) quasi-incompressible flows. It operates on *particle density distributions* as primary variables that represent the fraction of the total mass moving in a discrete direction. LBM has its roots in kinetic theory (as it is obtained as a special finite difference discretisation from the velocity-discrete Boltzmann equation), but it is applied to continuum mechanics (i.e., small Knudsen numbers) and the results satisfy the incompressible Navier-Stokes equations with second order of accuracy in space and first order in time.

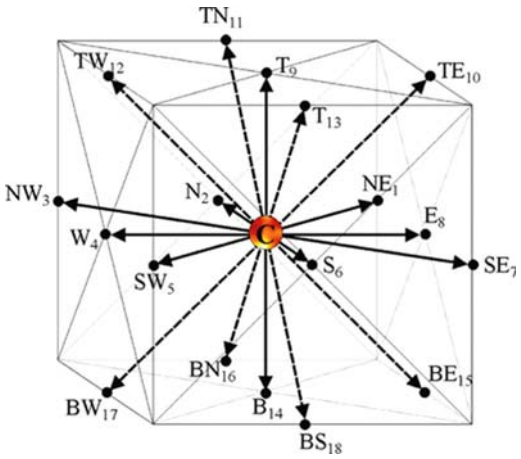
The lattice Boltzmann equation with *single-relaxation time* approximation of the collision operator (BGK model) reads with $i = 0, \ldots, N_e - 1$

$$f_i(\vec{x} + \vec{e}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{\Delta t}{\tau}\left[f_i(\vec{x}, t) - f_i^{\mathrm{eq}}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))\right], \quad (1)$$

where f_i is the particle density distribution corresponding to the discrete velocity direction $\vec{e}_i$. $\vec{x}$ and t are the discrete location (cell) and time. The time step Δt and the discrete velocity vectors $\vec{e}_i$ are chosen such that all distributions jump exactly from one cell to the next one during each time step. The deviation from the equilibrium state f_i^{eq} which only depends on the macroscopic density $\rho = \sum_0^{N_e-1} f_i$ and velocity $\vec{u} = \sum_0^{N_e-1} \vec{e}_i f_i / \rho$ is weighted with the collision time τ which is related to the viscosity of the fluid.

In our implementation, we use a 3D lattice with N_e=19 collocation points per cell (D3Q19 discretisation, see Figure 1), leading to a five-dimensional array of double precision values – $f(1{:}N_X, 1{:}N_Y, 1{:}N_Z, 1{:}N_e, 0{:}1)$ in column-major storage order (Fortran layout) – where the last index denotes the two grids between which we iterate to circumvent issues with data dependencies. N_X, N_Y, N_Z are the numbers of cells in each spatial direction. Thus, the values of direction x are consecutive in memory. In the following we will mainly use cubic geometries, i.e., we use $N = N_X = N_Y = N_Z$.

Figure 1 Discretisation model D3Q19 including 18 discrete velocity directions and the centre representing the fraction of non-moving particles (see online version for colours)



3 Cache oblivious LB algorithm

The basic idea of Frigo's Cache Oblivious Algorithm (COA, Frigo and Strumpen, 2005) to optimise the spatial and temporal locality of the data is an implicit space and time blocking of a space-time domain of unknowns. During the traversal of the domain, each unknown is updated by a finite difference stencil. As the lattice Boltzmann method is a stencil-based finite difference method, it is suitable for integration into the COA scheme, which results in a Cache-Oblivious Lattice Boltzmann Algorithm (COLBA).

The COA is based on a *divide-and-conquer strategy*. It divides the complete space-time domain into triangles, parallelograms and trapezoids and traverses the discretised space-time domain in a specific order optimised for cache reuse. For demonstration purpose of the update order, a 1D space (resulting in a 2D space-time domain) with a 3-point stencil (nearest neighbour) is assumed, i.e., the new value at each point x_i at time t depends on x_{i-1}, x_i, x_{i+1} at time $t-1$. The 3D lattice Boltzmann calculations are a straightforward extension of the 1D update scheme, resulting in a 4D space-time domain.

Figure 2 Illustration of a space cut in a 2D trapezoid consisting of 1D space region and 1D time region

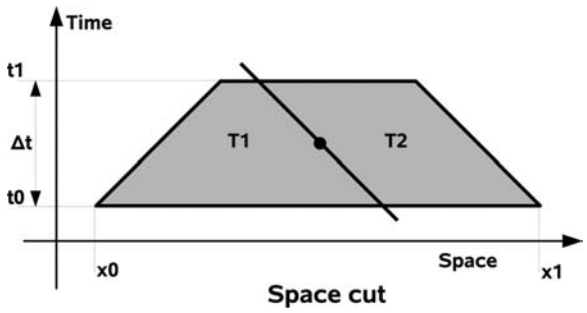
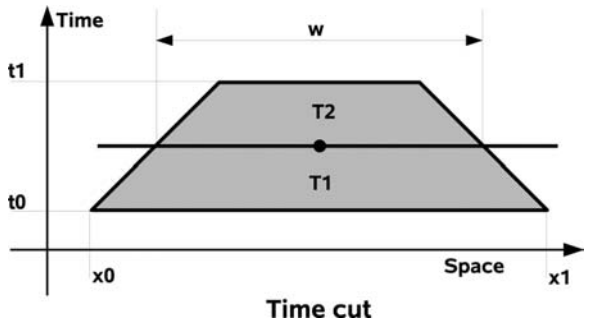


Figure 2 shows a 2D space-time trapezoidal region of the complete space-time domain. All points of the trapezoid are visited in an order that respects the stencil dependencies and decomposes the trapezoid recursively into smaller trapezoids either by *space* or *time cuts*. A *space cut* is done if the width of the trapezoid is at least twice its height by cutting the trapezoid along the line with slope -1 through the centre as illustrated in Figure 2. The slope of -1 ensures that data dependencies are preserved for stencil calculations involving nearest neighbours: The domain left of the cut always contains the values x_{i-1}, x_i, x_{i+1} at time $t-1$, which are needed to determine the value x_i at time t. If a space cut is not possible, a *time cut* is done along the horizontal line through the centre of a trapezoid (see Figure 3). Thus, in both cases, two smaller trapezoids $T1$ and $T2$ are created as shown in Figures 2 and 3. The same procedure is then recursively applied on these newly created trapezoids until no further cuts are possible. In this case, the LBM is applied twice: First on $T1$ and second on $T2$, which values next to the cut depend on results previously calculated in $T1$. Finally, the algorithm returns to the previous recursion level, i.e., the last (coarser) decomposition level. This traversal order is possible because no point in $T1$ depends on an updated (new) value of any point in $T2$. During the recursions, the size of the trapezoids decreases such that at a certain recursion level, the actual computational domain fits into the cache. Since – in contrast to explicit blocking techniques – no cache-size dependent parameters

are necessary to exploit the cache, the algorithm is characterised as *cache oblivious*. Due to the described blocking scheme in the space–time domain, it provides both good spatial and temporal locality.

Figure 3 Illustration of a time cut in a 2D trapezoid consisting of 1D space region and 1D time region



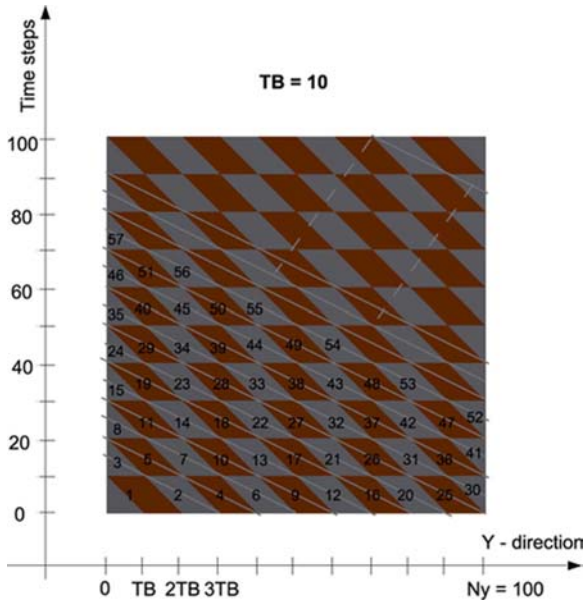
The extension to higher order spatial domains is straightforward: Space cuts are performed for each spatial direction separately before starting time cuts. Different stencils can also easily be incorporated by changing the slopes for the space cut.

4 Parallelisation

A first attempt to parallelise the COLBA in three spatial dimensions for shared memory systems is based on an extension of OpenMP. Basically, the space–time domain is decomposed in small subdomains which can be executed in parallel by applying the sequential algorithm. As a simple approach, we do not decompose the spatial domain in each spatial direction, but, instead, only along one spatial dimension (the y-direction as shown in Figure 4). Thus, we use a restricted 2D space-time domain with a decomposition into small subdomains as shown in Figure 4. In order to satisfy the stencil dependencies of the lattice Boltzmann kernel, the small domains are created in a similar way as the space cuts of the sequential COLBA on a 2D space–time domain by using parallelograms or triangles with a slope of -1 for the inclined edge. Due to the small size of the subdomains and to enhance the temporal locality, our C-based code uses an *array-of-structures* layout. Due to the data interdependencies induced by the stencils, all subdomains below and left of one of the diagonal thin lines in Figure 4 must have been completed already before the subdomains marked by the corresponding line can be computed. Hence, the propagating front of thin lines marks those subdomains which can be solved at a certain stage in parallel by individual, independent threads (one for each subdomain). Inside these small subdomains, a sequential COLBA algorithm as described in Section 3 is executed.

The number of parallel regions can be varied by changing the time steps solved within each small domain and the number of small domains in spatial dimension. For the workload (communication and load balance), it proved to be optimal to use sheared squares as parallelograms. Thus, we introduce a *time blocking factor (TB)* that describes the number of time steps and the spatial size in y-direction of a small domain. I.e., in Figure 4 the TB is set to 10. The TB is not necessary and not even reasonable in the serial version of COLBA, but only needed for domain decomposition in the parallel version. Currently, for simplicity the total number of time steps and the domain size have to be multiples of TB.

Figure 4 Parallel regions in a 2D time/space domain for $N_Y = 100$, $N_t = 100$ and $\mathrm{TB} = 10$ (see online version for colours)



For evaluation of the number of *effective parallel regions* (epr), equation (2) holds:

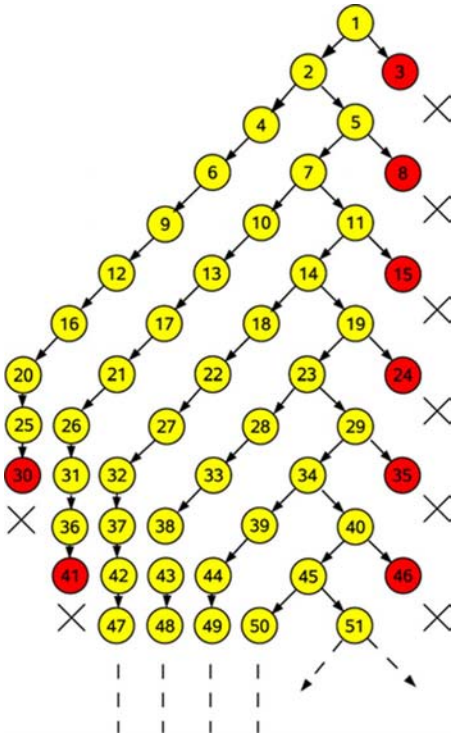
$$\mathrm{epr} = \frac{\min(N_Y, N_t)}{2 \cdot \mathrm{TB}}, \tag{2}$$

where N_Y and N_t denote the spatial extension in y-direction and the total number of time steps, respectively. The maximum number of threads that can run in parallel is $\mathrm{epr} + 1$ if the triangles at the boundary are computed by separated threads. Note that in this approach each 2D parallelogram in Figure 4 extends over the complete range (N_X, N_Z) of the remaining spatial dimensions (x, z) and extra care must be taken at the boundaries of the domain due to the different shapes of the subdomains and the boundary conditions to be performed.

The algorithm cannot be parallelised with standard OpenMP 2.5 techniques due to its recursive nature. However, recent versions of the Intel compilers feature an extension to the OpenMP 2.5 standard, called *task queue model* (Intel, 2005). Creating a task queue is performed via `#pragma intel omp taskq`. By `#pragma intel omp task`, it is then possible to add tasks to the queue, which are successively assigned to the available threads within a parallel region. In the algorithm, the parallel region is defined in the main routine, which calls the function that performs the recursion. Within this recursive function, the threads are spawned appropriately,

as shown in the tree diagram in Figure 5. In this tree, each node represents a small domain which is either a triangle or a parallelogram. The numbers inside the tree nodes correspond to the numbers of the small domains in the rectangular space-time region (see Figure 4). The long branches at the left side terminate when the computation of a row in the space-time domain is finished. When all branches are terminated, the complete space-time domain is solved.

Figure 5 Tree denoting order of traversal and spawning of threads (see online version for colours)



In order to guarantee correct data dependencies, and hence avoid that the computation of dependent domains starts before the calculation of a small domain is finished, locks on each small domain were introduced to synchronise the threads. Initially, all small domains are marked as locked. The threads first solve and then unlock the respective domain, and thus also have to check whether all dependent domains are unlocked before starting the computation of a small domain. In Figure 5, the tree shows that synchronisation has to occur on each level of the tree, e.g., the nodes 9 and 10 have to be solved before starting to compute the node 13.

5 Architectures

In this section, we briefly sketch the most important single-processor specifications of the architectures used in this report in Table 1 and the according performance values in Table 2. To our opinion, they perfectly reflect the ongoing transition from fast single-core processors to moderately clocked dual- or multi-core processors. As a baseline for our measurements we have chosen a classical single-core system which is the Intel Xeon/Nocona processor based on the well known Netburst architecture.

Table 1 Single processor specifications. Peak performance (Peak), capacity of largest (L2) cache level, and maximum memory bandwidth (MemBW). Dual-core based platforms are marked by asterisks (*). On the Woodcrest platform, the two cores share the 4 MB L2 cache

Platform	*Freq.* [GHz]	*Peak* [GFlop/s]	*L2* [MB]	*MemBW* [GB/s]
Intel Xeon/Nocona	3.2	6.4	1.0	5.3
AMD Opteron875*	2.2	4.4	1.0	6.4
Intel Xeon/Woodcrest*	2.66	10.6	4.0	10.6

Table 2 Single processor performance. Bandwidth as determined with the optimised STREAM benchmark (triad) (McCalpin, 2004) and maximum theoretical performance in MLUPs using the STREAM number together with the performance model proposed in Wellein et al. (2006b). Dual-core based platforms are marked by asterisks (*)

Platform	*STREAM* [GB/s]	*Max. MLUPs*
Intel Xeon/Nocona	3.0	6.6
AMD Opteron875*	3.5	7.7
Intel Xeon/Woodcrest*	4.2	9.2

By enhanced technical production processes, Intel was able to put two of these processors on a single silicon die, which are working completely independent but sharing the same (external) frontside bus (Intel Xeon 5000 series, codenamed 'Dempsey'). The same approach has been used by AMD earlier to set up a series of dual-core chips which hold two independently working Opteron CPUs. A substantial change has recently been introduced with the Intel Xeon 5100 processor series (codenamed 'Woodcrest') based on the Intel Core processor architecture, replacing the Intel Netburst design. Besides a completely new processor architecture these chips provide shared on-chip resources (shared L2 cache and fast data connections between the two L1 caches) for the two cores.

In the following, the terms 'CPU' and 'core' denote a classical 'full' processor. A dual-core chip consists of two processors, CPUs, or cores. In a multi-chip server, a single chip is placed into a 'socket', i.e., the widespread 4-way/socket Opteron servers can hold 4 processors if equipped with single-core and 8 if provided with dual-core chips.

The second column of Table 2 provides the effective attainable memory bandwidth for a single core as measured with the optimised STREAM triad benchmark (McCalpin, 2004). Based on this number, a maximum performance in terms of Mega Lattice site Updates Per second (MLUPs) can be estimated for the LBM kernel

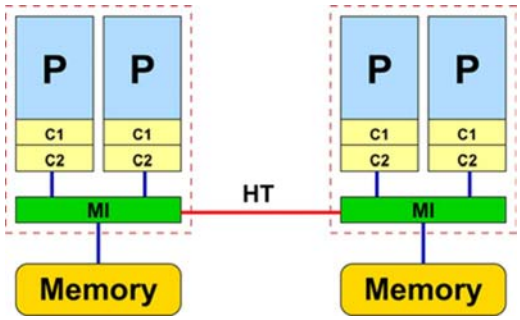
introduced in Wellein et al. (2006b) using equation (3) in Wellein et al. (2006b) and replacing the theoretical memory bandwidth with the STREAM numbers.

Our benchmark codes, the iterative and cache oblivious implementation, were compiled using the Intel Fortran Compiler for Intel EM64T-based applications in version 9.1.036 and the Intel C/C++ Compiler for Intel EM64T-based applications in version 9.1.043, respectively.

The AMD Opteron875 processors used in our benchmarks are 64-bit enabled dual-core versions of the well-known AMD Athlon designs, which maintain full IA32 compatibility. These are capable of performing a maximum of two double precision Floating Point (FP) operations (one multiply and one add) per cycle.

As outlined in Figure 6, the 4-way/socket Opteron server used here provides a separate memory path for each of the four dual-core chips, featuring a theoretical memory bandwidth of 25.6 GByte/s ($= 4 \times 6.4$ GByte/s) available for eight cores. Owing to the Hypertransport technology the Opteron server design allows easy scalability of memory bandwidth to the shared data space. However, Non-Uniform Memory Access (NUMA) effects, especially on Opteron systems where the cache-coherence protocol (ccNUMA) worsens the effect, must carefully be taken into account for a shared-memory implementation, e.g., LBM parallelisation based on OpenMP (Wellein et al., 2006a).

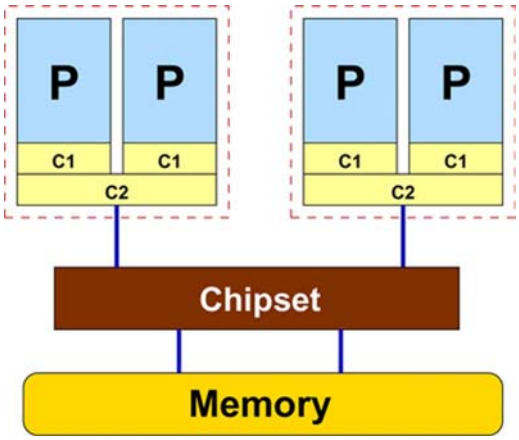
Figure 6 A dual-socket dual-core AMD Opteron system with processing unit (P), cache hierarchy (C1, C2), Hypertransport (HT) link, and integrated memory controller (MI) (see online version for colours)



Compared to the Intel Netburst design, the new Intel Core architecture introduces important changes which help to substantially improve the performance of numerically intensive codes. The most obvious one is that the capacity of the FP units has been doubled, i.e., the processor can execute two double precision FP add and two double precision FP mult operations in a single cycle, effectively doubling the peak performance at a given clock rate. Furthermore the instruction pipeline length has been reduced by a factor of roughly two and the cache latencies have been improved by almost the same factor, giving higher performance at short loop length.

In multi-chip/socket environments the Intel Xeon/Woodcrest chips, which are based on the Intel Core architecture, can use the 'Bensley' platform running at a maximum FSB of 1333 MHz. While in previous multi-processor systems (e.g., Intel Xeon/Nocona based systems) the single shared memory bus imposed a severe performance restriction for data intensive applications, on a Bensley system, the northbridge can in principle saturate two Front Side Buses (FSB) with up to 10.6 GB/s (FSB1333) each and still provides a Uniform Memory Access (UMA) for both dual-core chips (see Figure 7). However, the cache-coherency protocol can substantially reduce the sustainable bandwidth. In the benchmarks, a two socket system on a Bensley platform with 'Blackford' (X5000P) chipset and four 2.66 GHz Intel Xeon/Woodcrest (5150) CPUs was used.

Figure 7 A typical Intel 'Bensley' platform system with dual-core processors (P), private L1 and shared L2 caches (C1/C2). Dashed boxes symbolise sockets (see online version for colours)



6 Results

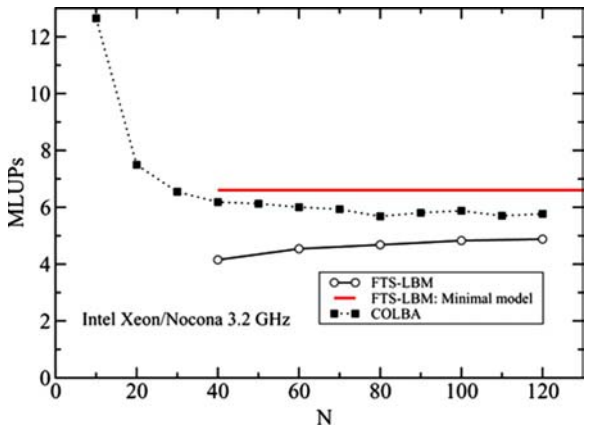
For the performance tests, COLBA computes the Lid Driven Cavity problem. In a 3D rectangular or cubic domain, all boundaries are solid walls while the wall at the top is moving with a constant speed. Based on the results from our previous work in Zeiser et al. (2008), in this paper we focus on the study of the performance on modern dual-core architectures with and without ccNUMA. For performance measurements of the LBM implementation without temporal optimisations, we have chosen the kernel presented in Wellein et al. (2006b), which has been shown to provide optimal data access. Since this approach iteratively performs each time step on the full grid, we refer to this as 'Full Time Step LBM' (FTS-LBM).

In our results the performance is given in MLUPS which is practical for measuring the performance of LBM, because this way it is possible to estimate the runtime of a real application from the domain size and the number of desired time steps. For LBM, even when using the same method, the total number of floating point operations per lattice site update can vary substantially depending on programmer's implementation and compiler. For our implementation of LBM, an analysis of the performance counters revealed a typical number of 200 Flops per lattice

site update, i.e., 5 MLUPS are equivalent to 1 GFlop/s of sustained performance.

As discussed in detail in Zeiser et al. (2008), the sequential performance of COLBA exceeds the FTS-LBM by 10–25%, although expectations based on the measured reduction in overall bus activity predicted a higher improvement. For most architectures, COLBA performance cannot even achieve the performance limit of the FTS-LBM algorithm which has been estimated in Table 2 and is shown as 'minimal model' in Figures 8–10 for intermediate and large domain sizes. One important reason for this is the fact that subdomains loaded consecutively into cache may be separated in main memory by large offsets. This shows up in a strongly increased number of Data Translation Look-aside Buffer (DTLB) misses for COLBA. Although the number of bus accesses is substantially reduced compared to FTS-LBM, COLBA generates four times more DTLB misses, which may cause substantial performance penalties. Consequently, COLBA cannot reasonably use the memory bus on a single core. Thus, a parallel implementation on multi-core based compute nodes is expected to be highly beneficial since running the code on two cores should give a considerable speedup even if they share the same path to memory.

Figure 8 Comparison of serial performance of COLBA and FTS-LBM over different domain sizes with minimal model as mentioned in Table 2 on Intel Xeon/Nocona (see online version for colours)



For the parallel implementation, in a first step the impact of the time blocking factor (TB) on the scalability of COLBA was studied. Figure 11 shows that epr (see equation 2) can give a very accurate approximation of the maximum number of threads that can run in parallel. E.g., for $TB = 10$ ($epr = 5$) a very good scalability for up to four threads is obtained, while there is only minor improvement in performance when using eight threads. Due to the task queue scheme, more than epr threads can still reduce the runtime slightly and compensate any runtime differences of threads. Consequently, choosing a small value of TB improves the scaling. However, the increased communication overhead introduced by the locking mechanism described in Section 4 may overcompensate the benefit of scalability, especially for small numbers of threads. The large benefit of parallel COLBA compared to the FTS-LBM by a factor of 1.5–2, although the serial performance differs only by 10–25%, arises from worse scaling due to a more costly boundary treatment in FTS-LBM.

Figure 9 Comparison of serial performance of COLBA and FTS-LBM over different domain sizes with minimal model as mentioned in Table 2 on AMD Opteron875 (see online version for colours)

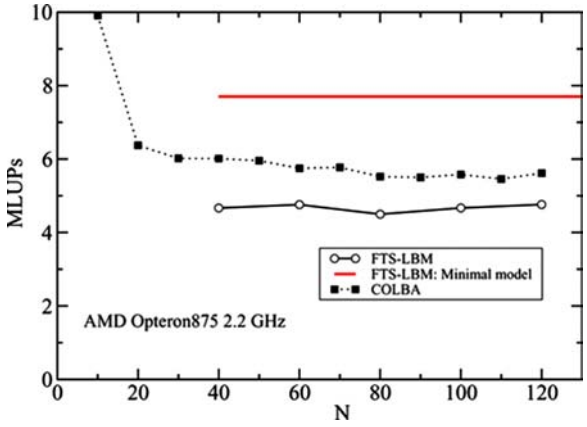
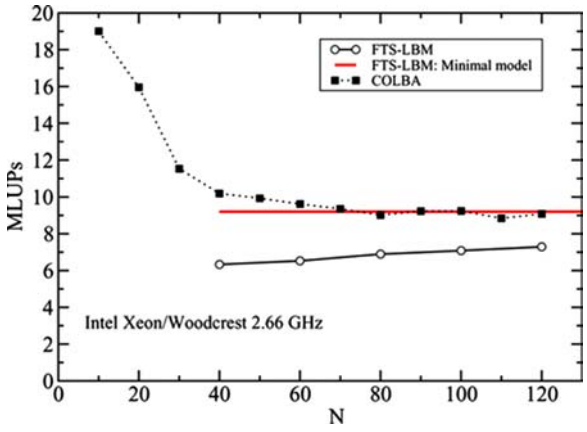


Figure 10 Comparison of serial performance of COLBA and FTS-LBM over different domain sizes with minimal model as mentioned in Table 2 on Intel Xeon/Woodcrest (see online version for colours)



A further study of different thread placements reveals the influence of the cache and memory architecture on multi-socket multi-core processor systems. For benchmarking, it is essential to define the mapping of threads or processors to the available sockets or cores. We used the *taskset* command to run the iterative LBM with either two threads on two cores of a single socket or on two separate sockets. Figure 12 depicts the performance of FTS-LBM on the Intel Xeon/Woodcrest system in comparison to a traditional single-core Nocona system.

For small domains, running two threads on the same socket is favourable since both cores access data that resides in the same shared cache. Two threads running on separated sockets have to exchange at least boundary values via the main memory system bus, which is very expensive compared to the pure cache accesses.

Moreover, cache coherence traffic impacts performance in this case, too.

Figure 11 Parallel performance of COLBA with a domain size of 100^3 ($N_t = 1000$) on AMD Opteron875 for different time blocking factors (TB). For comparison the results of FTS-LBM are given as well

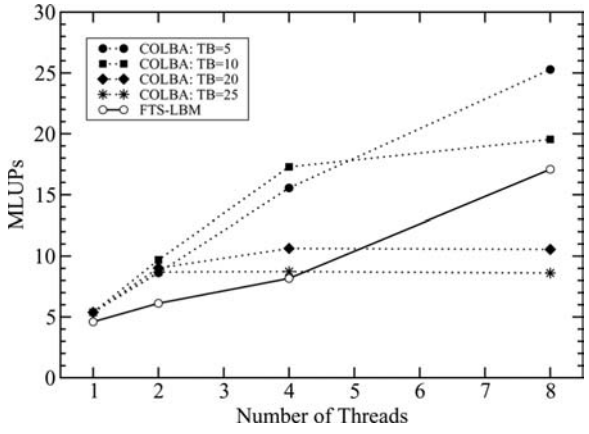
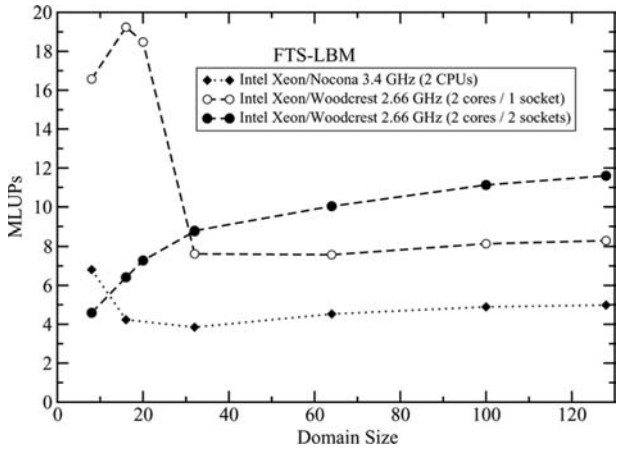


Figure 12 Parallel performance of iterative FTS-LBM with variable cubic domain size running two threads on either one or two sockets

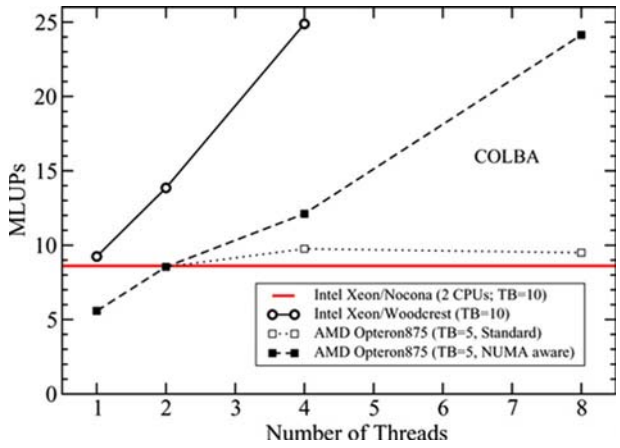


For large domains, distributing the threads on two sockets is more profitable. Owing to the new Intel bus architecture on the Woodcrest CPUs, the threads can use two memory busses instead of one on the Nocona system.

On Opteron systems ccNUMA effects play an important role. Figure 13 shows the different scaling of COLBA using a standard initialisation phase compared to a 'ccNUMA-aware' version. For this benchmark, always the smallest possible number of sockets was used, i.e., for two threads we used one socket, for four threads two, etc.

On ccNUMA systems, memory pages are allocated always in the memory that is nearest to the CPU which initialised these pages ('first touch' policy). Hence, when using a standard initialisation routine where thread 0 solely initialises the memory, data will be located in the local memory of the CPU thread 0 is running on. Thus, the algorithm scales well for up to two threads since both access their local memory. Scalability is worse for more than two threads because in this case several sockets access the same local memory of the CPU processing thread 0 via the hypertransport bus.

Figure 13 Parallel performance of COLBA for strong scaling with a domain size of 100^3 ($N_t = 1000$) on AMD Opteron875 and Intel Xeon/Woodcrest in comparison to a traditional Intel Xeon/Nocona architecture (see online version for colours)



A ccNUMA-aware initialisation strategy as introduced in Hager et al. (2006) should perform the 'first-touch' initialisation in exactly the same manner like the threads will operate on data later on. Due to the task queue model where no prescription can be made which thread works on which data and since each thread only computes a certain time block on each data, for COLBA there is no simple approach to realise a correct assignment of the data to the cores. However, applying a random distribution of the data on the four memory channels enhances the scalability sufficiently well (see curve denoted by 'NUMA aware' in Figure 13).

For comparison, as a baseline, the two-thread performance of a Nocona system is included in the Figure as well as the scaling on a two-socket Woodcrest system. Since 'first-touch' initialisation issue is special to NUMA architectures, the described effect currently appears not on Intel-based systems.

7 Conclusion

We have introduced our parallel cache oblivious blocking algorithm for the lattice Boltzmann method (COLBA), focusing on three spatial dimensions, and compared parallel COLBA performance with an 'optimal' iterative implementation using no temporal blocking techniques. We extended our studies of the parallel implementations on multi-core multi-socket systems and examined their scalability. The design of the memory system has tremendous effects on the parallel performance and scalability. While for small domain sizes pinning two threads on one dual-core socket is advantageous on an Intel Xeon/Woodcrest system, the use of separated sockets is more favourable for large domain sizes. Furthermore, we examined the influence of ccNUMA issues on

scalability of the parallel COLBA on an 4-way/socket Opteron875 system. A 'NUMA aware' implementation restores scalability in order to be comparable to the scaling on Intel systems.

Generally, our parallel COLBA shows very satisfying scaling behaviour on modern multi-core and multi-socket architecture. Due to the cache-oblivious approach and the minimal communication, scaling of COLBA outperforms the scaling of our 'optimal' iterative implementation, which makes Frigo's approach a promising technique for future many-core architectures. However, due to the recursive algorithm, for the lattice Boltzmann method, more complex boundary conditions are hard to implement and would compensate the advantages.

Acknowledgments

This work is financially supported by the Competence Network for Technical, Scientific High Performance Computing in Bavaria (KONWIHR). We gratefully acknowledge support from Intel.

References

Frigo, M., Leiserson, C. E., Prokop, H. and Ramachandran, S. (1999) 'Cache-oblivious algorithms', *40th Annual Symposium on Foundations of Computer Science, FOCS '99*, 17–18 October, 1999, New York, NY, USA.

Frigo, M. and Strumpen, V. (2005) 'Cache oblivious stencil computations', *International Conference on Supercomputing*, pp.361–366, ACM Press, Boston, MA.

Hager, G., Zeiser, T., Treibig, J. and Wellein, G. (2006) 'Optimizing performance on modern HPC systems: Learning from simple kernel benchmarks', in Krause, E., Shokin, Y., Resch, M. and Shokina, N. (Eds.): *Computational Science and High Performance Computing II: The 2nd Russian-German Advanced Research Workshop*, Stuttgart, Germany, 14–16 March, 2005, Notes on Numerical Fluid Mechanics and Multidisciplinary Design, Vol. 91, Springer, ISBN: 3-540-31767-8, DOI: 10.1007/3-540-31768-6_23.

Iglberger, K. (2003) *Cache Optimizations for the Lattice Boltzmann Method in 3D*, Bachelor's Thesis, Lehrstuhl für Informatik 10 (Systemsimulation), Universit ät Erlangen-Nürnberg.

Intel (2005) *Intel C++ Compiler Documentation*. Document number: 304967-005.

Kowarschik, M. (2004) *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*, PhD Thesis, Lehrstuhl für Informatik 10 (Systemsimulation), Advances in Simulation, No. 13, SCS Publ. House e.V., San Diego, Erlangen, ISBN: 3-936150-39-7.

McCalpin, J.D. (2004) *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, Technical report, University of Virginia.

Nitsure, A., Rüde, U., Iglberger, K., Wellein, G., Hager, G. and Feichtinger, C. (2006) 'Optimization of cache oblivious lattice Boltzmann method in 2D and 3D', in Becker, M. and Szczerbicka, H. (Eds.): *Frontiers in Simulation: Simulationstechnique – 19th Symposium in Hannover*, September 2006 (ASIM), pp.265–270, Erlangen, SCS Publishing House.

Pohl, T., Kowarschik, M., Wilke, J., Igelberger, K. and Rüde, U. (2003) 'Optimization and profiling of the cache performance of parallel lattice Boltzmann codes', *Parallel Process. Lett.*, Vol. 13, No. 4, pp.549–560.

Rivera, G. and Tseng, C.-W. (2000) 'Tiling optimizations for 3D scientific computations', *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, IEEE Computer Society, Dallas, Texas, Washington, DC, USA, p.32, ISBN: 0-7803-9802-5, http://doi.ieeecomputersociety.org/10.1109/SC.2000.10015

Schulz, M., Krafczyk, M., Tölke, J. and Rank, E. (2002) 'Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high performance computers', Breuer, M., Durst, F. and Zenger, C. (Eds.): *High Performance Scientific and Engineering Computing Proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, 12–14 March, 2001*, Vol. 21 of Lecture Notes in Computational Science and Engineering, pp.115–122, Springer, Berlin.

Stockman, H.W. and Glass, R.J. (1998) 'Accuracy and computational efficiency in 3D dispersion via lattice-Boltzmann: Models for dispersion in rough fractures and double-diffusive fingering', *IJMPC International Journal of Modern Physics C*, Vol. 9, No. 8, pp.1545–1557.

Succi, S. (2001) *The Lattice Boltzmann Equation – for Fluid Dynamics and Beyond*, Clarendon Press.

Wellein, G., Lammers, P., Hager, G., Donath, S. and Zeiser, T. (2006a) 'Towards optimal performance for lattice Boltzmann applications on terascale computers', in Deane, A., Brenner, G., Emerson, D., McDonough, J., Periaux, J., Tromeur-Dervout, D., Ecer, A. and Satofuka, N. (Eds.): *Parallel Computational Fluid Dynamics: Theory and Applications, Proceedings of the 2005 International Conference on Parallel Computational Fluid Dynamics*, 24–27 May, College Park, MD, USA, pp.31–40, Elsevier B.V., Amsterdam, Netherlands.

Wellein, G., Zeiser, T., Donath, S. and Hager, G. (2006b) 'On the single processor performance of simple lattice Boltzmann kernels', *Comput. and Fluids*, Vol. 35, pp.910–919.

Zeiser, T., Wellein, G., Nitsure, A., Iglberger, K. and Hager, G. (2008) 'Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method', *Progress in Computational Fluid Dynamics*, Geneve, Inderscience Publishers, Switzerland, Vol. 8, Nos. 1–4, pp.179–188, DOI: 10.1504/PCFD.2008.018088.