Parallel Processing Letters Vol. 13, No. 4 (2003) 549–560

OPTIMIZATION AND PROFILING OF THE CACHE PERFORMANCE OF PARALLEL LATTICE BOLTZMANN CODES

THOMAS POHL, MARKUS KOWARSCHIK, JENS WILKE,
KLAUS IGLBERGER, ULRICH RÜDE

System Simulation Group, Dept. of Computer Science, University of Erlangen-Nuremberg,
Cauerstraße 6, D-91058 Erlangen, Germany
{Pohl,Kowarschik,Wilke,Iglberger,Ruede}@cs.fau.de

Received February 16, 2003
Revised September 26, 2003
Communicated by Harald Kosch

ABSTRACT

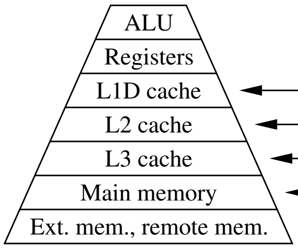
When designing and implementing highly efficient scientific applications for parallel computers such as clusters of workstations, it is inevitable to consider and to optimize the single-CPU performance of the codes. For this purpose, it is particularly important that the codes respect the hierarchical memory designs that computer architects employ in order to hide the effects of the growing gap between CPU performance and main memory speed. In this article, we present techniques to enhance the single-CPU efficiency of lattice Boltzmann methods which are commonly used in computational fluid dynamics. We show various performance results for both 2D and 3D codes in order to emphasize the effectiveness of our optimization techniques.

Keywords: lattice Boltzmann method, cache, performance, code optimization, profiling

1. Introduction

In order to enhance the performance of any parallel scientific application, it is important to focus on two related optimization issues. Firstly, it is necessary to minimize the parallelization overhead itself. These efforts commonly target the selection of appropriate load balancing strategies as well as the minimization of communication overhead by hiding network latency and bandwidth limitation. Secondly, it is necessary to exploit the individual parallel resources as efficiently as possible; e.g., by achieving as much performance as possible on each CPU in the parallel environment. This is especially true for distributed memory systems found in computer clusters based on off-the-shelf workstations communicating via fast networks. We concentrate on the second optimization issue in this article.

In order to mitigate the effects of the growing gap between theoretically available processor speed and main memory performance, today's computer architectures are typically based on *hierarchical memory designs*, involving CPU registers, several levels of cache memories (caches), and main memory [10]. Remote main memory



Capacity	Bandwidth	Latency
16 KB	16 GB/s	1 cycle
256 KB	32 GB/s	5+ cycles
1.5/3 MB	32 GB/s	12+ cycles
2 GB	6.4 GB/s	>100 cycles

Figure 1: Memory architecture of a workstation based on an Intel Itanium2 CPU with three levels of on-chip cache [12].

and external memory (e.g., hard disk drives) can be considered as the slowest components in any memory hierarchy. Fig. 1 illustrates the memory architecture of a current high performance workstation.

Efficient execution in terms of work units per second can only be obtained if the codes exploit the underlying memory design. This is particularly true for numerically intensive codes. Unfortunately, current compilers cannot perform highly sophisticated code transformations automatically. Much of this optimization effort is therefore left to the programmer [8].

Generally speaking, efficient parallelization and cache performance tuning can both be interpreted as *data locality optimizations*. The underlying idea is to keep the data to be processed as close as possible to the corresponding ALU. From this viewpoint, cache optimizations form an extension of classical parallelization efforts.

Research has shown that the cache utilization of iterative algorithms for the numerical solution of systems of linear equations can be improved significantly by applying suitable combinations of *data layout optimizations* and *data access optimizations* [3,6]. The idea behind these techniques is to enhance the *spatial locality* as well as the *temporal locality* of the code [10]. Similar work focuses on other algorithms of numerical linear algebra [15] and hardware-oriented FFT implementations [7], for example. An overview of cache optimization techniques for numerical algorithms can be found in [13]. Our current work concentrates on improving the cache utilization of parallel implementations of the *lattice Boltzmann method (LBM)*, which represents a particle-based approach towards the numerical simulation of problems in computational fluid dynamics (CFD) [5,18].

This article, which is an extended version of [17], is structured as follows. Section 2 contains a brief introduction to the LBM. Section 3 presents code transformation techniques to enhance the single-CPU performance of the LBM and introduces a *compressed grid storage* technique. Performance results and cache profiling information for LBM implementations in 2D and in 3D on various platforms are presented in Section 4. Finally, we conclude in Section 5.

2. The Lattice Boltzmann Method

It is important to note that we provide only a brief description of the LBM in this section, since the actual physics behind this approach are not essential for the

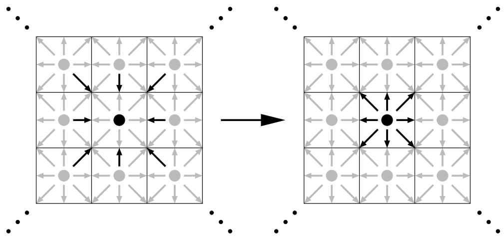


Figure 2: Representation of the 2D LBM updating a single lattice site by a stream operation (left) and a subsequent collide operation (right).

application of our optimization techniques.

The usual approach towards solving CFD problems is based on the numerical solution of the governing partial differential equations, particularly the Navier-Stokes equations. The idea behind this approach is to discretize the computational domain using finite differences, finite elements or finite volumes, to derive algebraic systems of equations, and to solve these systems numerically [9].

In contrast, the LBM is a particle-oriented technique, which is based on a microscopic model of the moving fluid particles [18]. Using $\vec{x}$ to denote the position in space, $\vec{u}$ to denote the particle velocity, and t to denote the time parameter, the so-called *particle distribution function* $f(\vec{x}, \vec{u}, t)$ is discretized in space, velocity, and time. This results in the computational domain being regularly divided into cells (so-called *lattice sites*). The current state of each lattice site is defined by an array of floating-point (FP) numbers that represent the distribution functions w.r.t. to the discrete directions of velocity.*

The LBM proceeds as follows. In each time step, the entire grid (lattice) is traversed, and the distribution function values at each site are updated according to the states of its neighboring sites in the grid at the previous discrete point in time. This update step consists of a *stream* operation, where the corresponding data from the neighboring sites are retrieved, and a *collide* operation, where the new distribution function values at the current site are computed according to a suitable model of the microscopic behavior of the fluid particles, preserving hydrodynamic quantities such as mass density, momentum density, and energy. Usually, this model is derived from the *Boltzmann equation*

$$\frac{\partial f}{\partial t} + \langle \vec{u}, \nabla f \rangle = \frac{1}{\lambda}\left(f - f^{(0)}\right) \quad ,$$

which describes the time-dependent behavior of the particle distribution function f. In this equation, $f^{(0)}$ denotes the equilibrium distribution function, λ is the relaxation time, $\langle ., . \rangle$ denotes the standard inner product, and ∇f denotes the gradient of f w.r.t. the spatial dimensions [18].

Fig. 2 shows the update of an individual lattice site in 2D for a so-called D2Q9 LBM model, since there are nine distribution functions per lattice site. The grid on

*In contrast to the *lattice gas* approach [18] where hexagonal grids are more common, we focus on orthogonal grids in 2D and in 3D, since they are almost exclusively used for the LBM.

the left illustrates the source grid corresponding to the previous point in time, the darker arrows represent the particle distribution function values being read. The grid on the right illustrates the destination grid corresponding to the current point in time, the dark arrows represent the new particle distribution function values; i.e., after the collide operation.

Note that, in every time step, the lattice may be traversed in any order since the LBM only accesses data corresponding to the previous point in time to compute new distribution function values. From an abstract point of view, the structure of the LBM thus parallels the structure of an implementation of Jacobi's method for the iterative solution of linear systems on structured meshes. In particular, the time loop in the LBM corresponds to the iteration loop in Jacobi's method.

3. Optimization Techniques for Lattice Boltzmann Codes

3.1. Fusing the stream and the collide operation

A naive implementation of the LBM would perform two entire sweeps over the whole data set in every time step: one sweep for the stream operation, copying the distribution function values from each lattice site into its neighboring sites, and a subsequent sweep for the collide operation, calculating the new distribution function values at each site.

A first step to improve performance is to combine the streaming and the collision step. The idea behind this so-called *loop fusion* technique is to enhance the temporal locality of the code and thus the utilization of the cache [1]. Instead of passing through the data set twice per time step, the fused version retrieves the required data from the neighboring cells and immediately calculates the new distribution function values at the current site, see again Fig. 2. Since this tuning step is both common and necessary for all subsequent transformations, we consider this fused version as our starting point for further cache optimizations in 2D and 3D.

3.2. Data Layout Optimizations

Accessing main memory is very costly compared to even the lowest cache level. Hence, it is essential to choose a memory layout for the implementation of the LBM which allows to exploit the benefits of hierarchical memory architectures.

Since we need to maintain the grid data for any two successive points in time, our initial storage scheme is based on two arrays of records. Each of these records stores the distribution function values of an individual lattice site. Clustering the distribution function values is a reasonable approach in 2D and in 3D since the smallest unit of data to be moved between main memory and cache is a *cache block* which typically contains several data items that are located adjacent in memory.

In the following, we present two data layout transformations that aim at further enhancing the spatial locality of the LBM implementation; *grid merging* and *grid compression*.

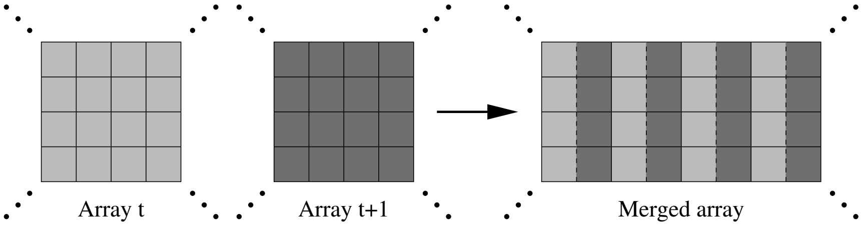


Figure 3: The two 2D arrays which store the grid data at time t and time $t+1$, respectively, are merged into a single array.

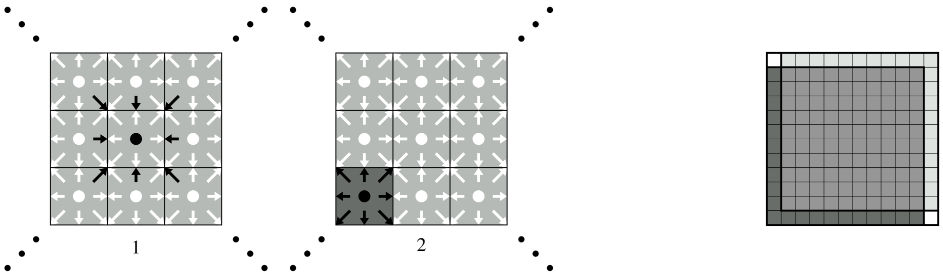


Figure 4: Fused stream-and-collide operation for the cell in the middle (left, Grids 1 and 2), layout of the two overlayed grids after applying the grid compression technique to the 2D LBM (right).

Grid merging. During a fused stream-and-collide step it is necessary to load data from the grid (array) corresponding to time t, to calculate new distribution function values, and to store these results into the other grid corresponding to time $t+1$.

Due to our initial data layout (see above) the first two steps access data which are tightly clustered in memory. Storing the results, however, involves the access of memory locations that can be far away from each other in memory. In order to improve spatial locality, we introduce an interleaved data layout where the distribution function values of each individual site for two successive points in time are kept next to each other in memory. This transformation is commonly called *array merging* [13]. Fig. 3 illustrates the application of this technique for the 2D case, while it can be applied analogously in 3D [11,16].

Grid compression. The idea behind this data layout is both to save memory and to increase spatial locality. For the ease of presentation, we concentrate on the 2D LBM in the description below, see also [16]. Like in the grid merging case, however, the extension of this technique to 3D is straightforward [11].

In an implementation of the 2D LBM, nearly half of the memory can be saved by exploiting the fact that only the data from the eight neighboring cells are required to calculate the new distribution function values at any regular site of the grid[†] It is therefore possible to overlay the two grids for both points in time, introducing a diagonal shift of one row and one column of cells into the stream-and-collide operation. The direction of the shift then determines the update sequence in each time step since we may not yet overwrite those distribution function values that are still required in subsequent lattice site updates.

In Fig. 4, the light gray area contains the values for the current time t. The two figures on the left illustrate the stream-and-collide operation for a single cell: the values for time $t+1$ will be stored with a diagonal shift to the lower left. Therefore, the stream-and-collide sweep must start with the cell located in the lower left corner of the grid. Consequently, after one complete sweep, the new values (time $t+1$) are shifted compared to the previous ones (time t). For the subsequent time step, however, the sweep must start with the cell in the upper right corner. The data for time $t+2$ must then be stored with a shift to the upper right. After two successive sweeps, the memory locations of the distribution functions are the same as before. This alternating scheme is illustrated in the right picture of Fig. 4.

3.3. Data Access Optimizations

Data access optimizations change the order in which the data are referenced in the course of the computation, while respecting all data dependencies. Our access transformations for implementations of the LBM in 2D and 3D are based on the *loop blocking (loop tiling)* technique. The idea behind this general approach is to divide the iteration space of a loop or a loop nest into blocks and to perform as much computational work as possible on each individual block before moving on to the next block. Blocking an individual loop means replacing it by an outer loop, which controls the movement of the block, and an inner loop, which traverses the block itself. If the size of the blocks is chosen appropriately, loop blocking can significantly enhance cache utilization and thus yield significant application speedups [1,8,13].

In the following, we present a selection of blocking approaches in order to increase the temporal locality of the LBM implementations in 2D and 3D. Each blocking technique takes advantage of the stencil-based memory access exhibited by the LBM. Because of this local operation, a grid site can be updated to time $t+1$ as soon as all the sites it depends on have been updated to time t. We use the term *n-way blocking* to indicate that n of the nested loops have been blocked, resulting in n additional loops.

The 2D case. Fig. 5 illustrates an example of *1-way blocking* in 2D, where two successive time steps are blocked into a single pass through the grid. The areas surrounded by thick lines mark the blocks of cells to be updated next. White cells have been updated to time t, light gray cells to time $t+1$, and dark gray cells to

[†]For the sake of simplicity, we focus on regular (interior) sites and omit the description of how to treat boundary sites and obstacle sites separately.

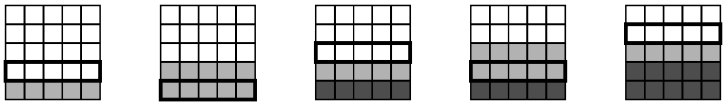


Figure 5: 1-way blocking technique for the 2D LBM.

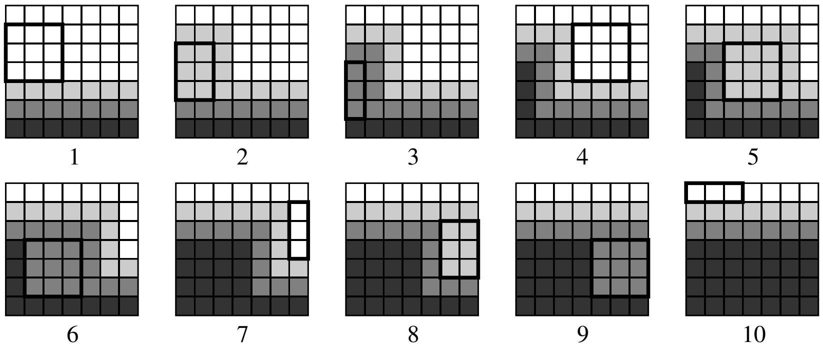


Figure 6: 3-way blocking technique for the 2D LBM.

time $t+2$. It can be seen that, in the second grid, all data dependencies of the bottom row are fulfilled, and that it can therefore be updated to time $t+2$, which is shown in the third grid. This is performed repeatedly until all cells have been updated to time $t+2$. The downside to this method is that even two rows may already contain too much data to fit into cache. In this case, no performance gain will be observed.

Fig. 6 illustrates an example of *3-way blocking* in 2D, in which a 3×3 block of cells is employed. Three successive time steps are performed during a single pass over the grid. Since the amount of data contained in the 2D block is independent of the grid size, it will always (if the block size is chosen appropriately) fit into the highest possible cache level. Again, the areas surrounded by thick lines mark the blocks of cells to be updated next.

A cell can be updated to time $t+1$ only if all eight adjacent cells correspond to time t. Consequently, the block to be processed moves diagonally down and left to avoid violating data dependencies. Obviously, special handling is required for those sites near grid edges which cannot form a complete 3×3 block. In these cases, parts of the blocks outside the grid are simply chopped off. Assume that we have arrived at the state of Grid 1. Grids 2 to 4 then demonstrate the handling of the next block of cells: Grid 2 shows that the marked block from Grid 1 has been updated to time $t+1$, Grid 3 shows the update of the marked block from Grid 2 to time $t+2$, and Grid 4 shows that the marked block from Grid 3 has been updated to time $t+3$. Grids 5 to 7 illustrate the handling of the next block, and so on. This is the common case occurring for all interior 3×3 blocks which do not require any special boundary handling.

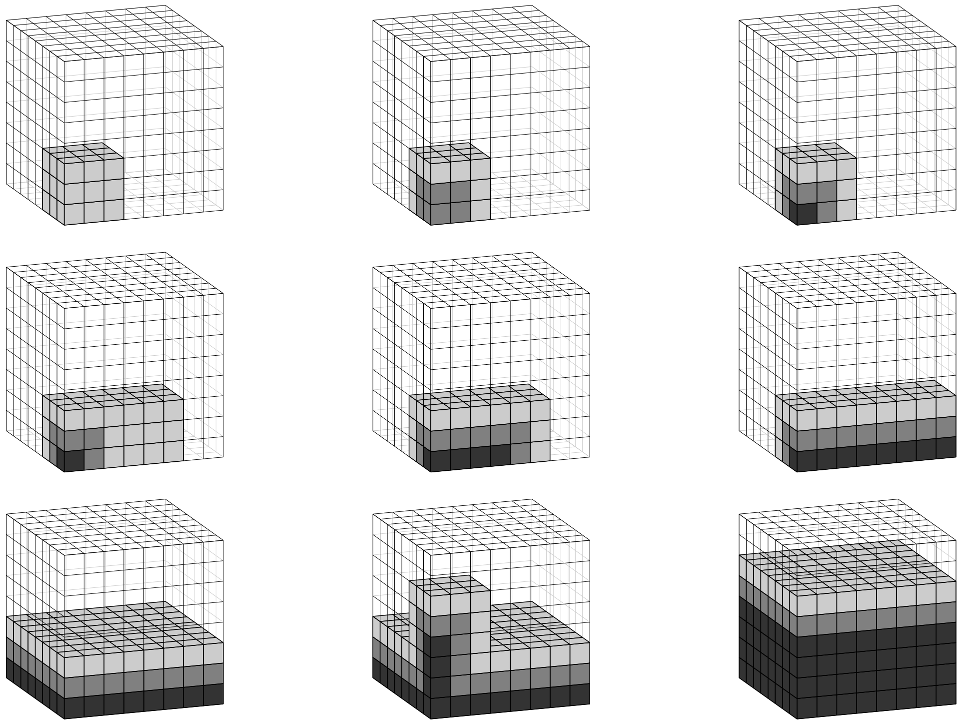


Figure 7: 4-way blocking technique for the 3D LBM.

The 3D case. Since the data sizes of individual layers and even rows of large 3D grids easily exceeds current cache capacities,[‡] we have implemented a 3-way and a 4-way blocking approach. In our 3-way blocking approach, the three loops along the spatial dimensions are blocked, while the time loop remains unchanged.

Fig. 7 illustrates the more involved 4-way blocking technique. For the ease of presentation, only a small block of $3 \times 3 \times 3$ grid cells is used, and three successive time steps are blocked into a single pass through the grid. The 4-way blocking approach in 3D parallels the 3-way blocking approach in 2D, cf. Fig. 6.

3.4. Software Design

In general, the combination of code flexibility and efficiency plays a central role in software engineering. On one hand, codes should be designed in a flexible and modular fashion, such that they can easily be reused, ported, extended, and maintained. On the other hand, it is often unavoidable to introduce highly system-specific optimizations in order to achieve acceptable runtime performance on the individual target platforms.

[‡]For a D3Q19 LB model with double precision FP values, the size of a single 100^2 grid layer is already about 1.6 MB. Storing an entire 100^3 grid using grid compression requires about 172 MB.

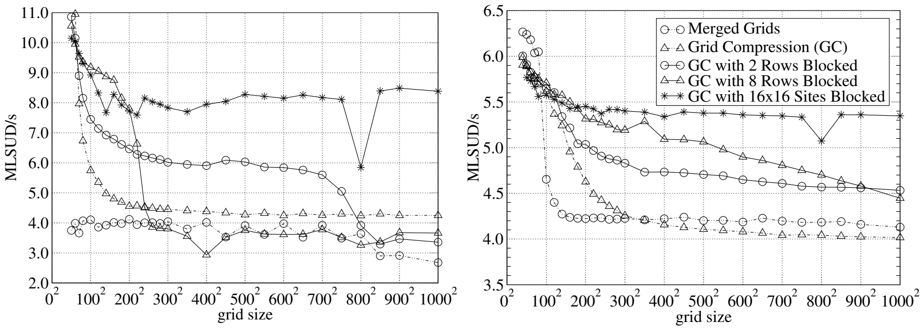


Figure 8: Performance of the 2D code in millions of lattice site updates per second (MLSUD/s) on machines based on an AMD Athlon XP 2400+ (left) and on an Intel Itanium2 (right), respectively, for various grid sizes.

In order to achieve both seemingly conflicting goals, our codes are based on the application of various *meta-programming techniques* such as automated code generation using macro expansion mechanisms, for example. In particular, this approach allows the data layout to be almost completely decoupled from the data access pattern; i.e., the blocking approach. This means that each of our data access transformations from Section 3.3 can easily be combined with either of the two data layout transformations from Section 3.2, cf. [11,16].

4. Performance Results

In order to benchmark the various implementations of the LBM, a well-known problem in fluid dynamics, known as the *lid-driven cavity*, has been used [9].

Performance. Fig. 8 demonstrates the performance of five different implementations of the 2D LBM in ANSI C++ on two different machines[§] for various grid sizes. The simple implementation involving a merged source and destination grid is the slowest, the implementation using the layout based on grid compression is slightly faster. Two implementations of 1-way blocking are shown, one with two time steps blocked, and one with eight. Initially, they show a significant increase in performance. For large grids, however, performance degrades since the required data cannot fit into even the lowest level of cache. This effect is more dramatic on the AMD processor since it has only 256 kB of L2 and no L3 cache, whereas the Intel CPU has 1.5 MB of L3 cache on-chip. Finally, the 2D blocked implementation shows a significant increase in performance for all grid sizes. It should be noted that each blocked implementation based on the grid merging layout performs worse

[§] We use an AMD Athlon XP 2400+ based PC (2.0 GHz), Linux, gcc 3.2.1, an Intel Itanium2 based HP zx6000 (900 MHz), Linux, Intel ecc V7.0, an AMD Opteron based PC (1.6 GHz), Linux, gcc 3.2.2, and a DEC Alpha A21164 based Digital PWS 500au (500 MHz), Compaq Tru64 UNIX, Compaq cc V6.1. Aggressive compiler optimizations have been enabled in all experiments.

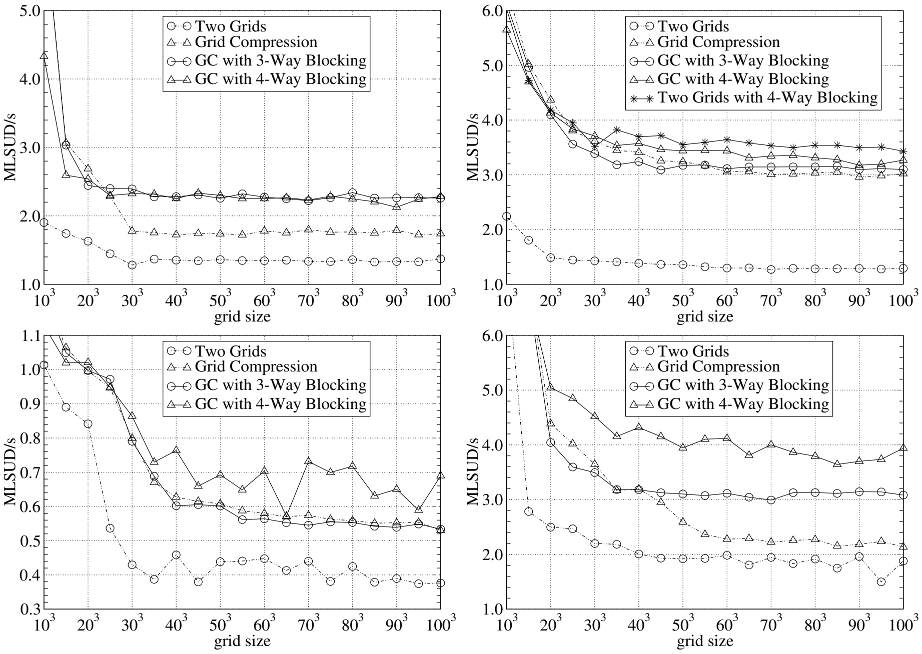


Figure 9: Performance of the 3D code in MLSUD/s on machines based on an AMD Athlon XP 2400+ (upper left), an Intel Itanium2 (upper right), a DEC Alpha A21164 (lower left), and an AMD Opteron (lower right), respectively, for various grid sizes.

than its counterpart based on grid compression. Therefore, no performance results for blocked codes using the grid merging technique are shown.

While the MLSUD/s rates in 3D are generally lower than in 2D, the speedup results for the 3D case are qualitatively similar, see Fig. 9. In particular, the DEC Alpha and the AMD Opteron systems show significant performance gains. Although the DEC Alpha machine is outperformed by all current architectures and must be considered outdated, its large L3 cache (4 MB) represents future trends concerning the increase of cache capacities. Results for the grid merging technique are omitted due to the better performance of the grid compression approach. Only on the Intel Itanium2 machine, the 4-way blocked code employing two separate grids outperforms its counterpart based on grid compression.

Cache Behavior. Fig. 10 demonstrates the behavior of the L1 and L2 cache of the AMD Athlon XP for the different implementations of the 2D LBM. These results have been obtained by using the profiling tool *PAPI* [4]. When compared to Fig. 8 it can be seen that there is a strong correlation between the number of cache misses and the performance of the code. The correlation between the performance drop of the two 1D blocked implementations and their respective rise in L2 cache

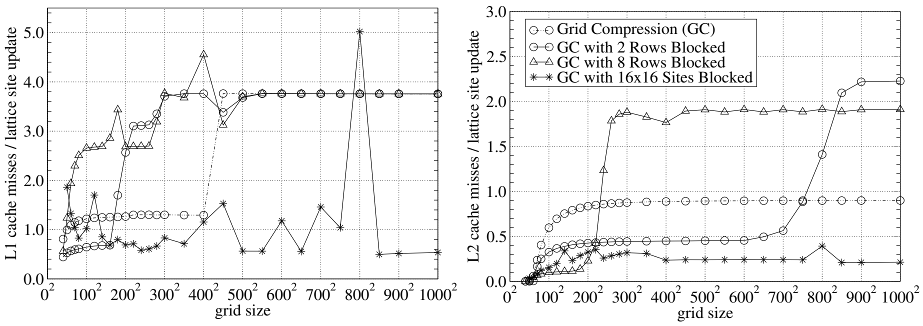


Figure 10: Cache behavior of the AMD Athlon XP for different 2D LBM codes.

Event (in 10^6)	Two grids, no blocking	GC, no blocking	Two grids, 4-way blocking	GC, 4-way blocking
L1 (8 kB) miss	272.7	276.8	278.3	294.4
L2 (96 kB) miss	407.0	413.4	308.1	272.5
L3 (4 MB) miss	121.7	45.8	43.9	25.8
TLB miss	1.5	1.1	8.6	4.9

Table 1: Hardware event counts for different 3D codes measured with DCPI [2] on a DEC Alpha A21164 architecture, problem size 100^3.

misses is especially dramatic. Additionally, Fig. 10 reveals the cause of the severe performance drop exhibited by the 2D blocked implementation at a grid size of 800^2. The high number of L1 misses are *conflict misses* [14]. They occur when, due to the limited set associativity of the cache, large amounts of data are mapped to only a few cache lines and therefore cause *cache thrashing* [8].

Table 1 contains cache profiling data for the DEC Alpha A21164 architecture. These numbers show that, due to the enormous data set sizes occurring in 3D, our optimization techniques mainly improve the utilization of the large L3 cache, while the L2 miss rate decreases by only about 30%. However, the TLB miss rate increases by about a factor of 3 due to the fact that our blocking approaches require higher numbers of virtual pages to be accessed alternatingly [10].

5. Conclusions

Due to the still widening gap between CPU and memory speed, hierarchical memory architectures will continue to be a promising optimization target. CPU manufacturers have already announced new generations of processors with several megabytes of on-chip cache in order to hide the slow access to main memory.

We have demonstrated the importance of considering the single-CPU performance before using parallel computing methods in the framework of a CFD code based on the LBM in 2D and in 3D. By applying optimizing code transformations and thus exploiting the benefits of hierarchical memory architectures of current CPUs, we have obtained speedup factors of up to more than 3 on various machines.

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2001.

[2] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 1–14, St. Malo, France, 1997.

[3] F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures. In *Proc. of the Int. Conf. on Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, NV, USA, 1998.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. of High Performance Computing Applications*, 14(3):189–204, 2000.

[5] S. Chen and G.D. Doolen. Lattice Boltzmann Method for Fluid Flow. *Annual Reviews of Fluid Mechanics*, 30:329–364, 1998.

[6] C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.

[7] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, USA, 1998.

[8] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.

[9] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics*. SIAM, 1998.

[10] J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.

[11] K. Iglberger. Cache Optimizations for the Lattice Boltzmann Method in 3D. Lst. für Informatik 10, Inst. für Informatik, University of Erlangen-Nuremberg, Germany, September 2003. *http://www10.informatik.uni-erlangen.de/dime*.

[12] Intel Corporation. *Intel Itanium2 Processor Reference Manual*, 2002. Document Number: 251110–001.

[13] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, pages 213–232. Springer, 2003.

[14] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[15] R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proc. of the ACM/IEEE Supercomputing Conf.*, Orlando, FL, USA, 1998.

[16] J. Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D. Lst. für Informatik 10, Inst. für Informatik, University of Erlangen-Nuremberg, Germany, February 2003. *http://www10.informatik.uni-erlangen.de/dime*.

[17] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde. Cache Performance Optimizations for Parallel Lattice Boltzmann Codes. In *Proc. of the Euro-Par 2003 Conf.*, volume 2790 of *LNCS*, pages 441–450. Springer, 2003.

[18] D.A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*, volume 1725 of *LNM*. Springer, 2000.