FRIEDRICH-ALEXANDER-UNIVERSITÄT
ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Bachelor Thesis

Improving computational efficiency of lattice Boltzmann methods on complex geometries

Johannes Habich

Improving computational efficiency of lattice Boltzmann methods on complex geometries

Johannes Habich

Bachelor Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dr. Georg Hager
	Dr. Gerhard Wellein,
	Dipl.-Ing. Thomas Zeiser
Bearbeitungszeitraum:	Juli 2005 - Februar 2006

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 14.02.2006 .....

Abstract

The lattice Boltzmann method has evolved over the past decade to a common technique in the field of computational fluid dynamics. The basic principle of cellular-automata allows an easy simulation of discrete surfaces and implementation of different boundary conditions. To meet the numerical requirements of continuous surfaces the enhanced Bouzidi bounce-back with interpolation was implemented. As the LBM requires a fine grained grid base, thus high memory usage, a particular emphasis was put on performance issues and performance optimization. To meet the increased obstacle to fluid ratio of unsteady surfaces, an unstructered list format was introduced, which turned out to be insensitive to the obstacle fluid ratio in terms of performance. The unstructured list algorithm with enhanced spatial data locality, finally performed nearly equal to the full matrix implementation. To exploit the growing parallelism in state-of-the-art compute nodes, a shared memory parallelization approach was implemented.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The lattice Boltzmann method (LBM) has been established in several new fields of CFD and a growing amount of classical solved problems is ported to this new numerical method. The relatively simple core structure enables an easy enhancement of implementation details and numerics. Analysis of the CFD advantages of LBM in comparison to the classical Navier-Stokes solvers (e.g. [20]) as well as detailed performance behavior ([19]) have been done. The aim of this thesis is to extend the work of Iglberger [11] and Donath [8] from empty channel layouts to complex geometries, utilizing inclined channels and sphere flow objects. Therefor a more sophisticated model of the boundary conditions called Bouzidi's et al. [6] bounce-back with interpolation, based on the results of the preceding work by Zschaek [21] is used.

The shortcoming of the simple bounce-back routine without interpolation is the implicit generated staircase, which results on the distinction between only fluid and obstacle cells. Especially with the above mentioned skewed channel and sphere obstacles in the fluid or in general any curved surface, the accuracy of the boundary simulation is degraded from second order accuracy to first order (e.g. [21]). The advanced algorithm has an interpolation [6] improved bounce-back routine which does additional calculations to mixed fluid obstacle cells. However these calculations as well as the storing of these mixed cells require extra computing power and reduce the performance gain of classical optimization techniques. The algorithm as well as the data layout with a full matrix, fits best to open channel simulation. However the issue of storing more and more obstacles with an growing incline and a growing sphere, emerges. In order to reduce the amount of unnecessary stored cells the newly implemented unstructured list layout is introduced.

To provide comparable and interpretable results, the test case scenarios include different inclined channels from fully open to half open. Additionally a sphere is placed inside the channel. For future applications especially blood vessel flow simulations and porous media, thus far more complex surfaces then inclined channels, highly efficient data storage models, high-performance algorithms and accurate numerical methods are evaluated.

The LBM method is a direct solver, which requires extensive computer resources. Therefor the code was optimized to get the most performance out of state of the art computing systems. At the present time the efforts for more computing power have reached a new stage of development. Increasing the clock rate was the appliance to satisfy the need for more computing power in the last decades. However the limit has been reached with about 3.8 GHz in today's high end processors, which could be hardly cooled anymore. The next stage appliance is more cores in one chip. The actual development is talking about dual and quad cores but also about multi and many cores. Regarding the growing amount of parallelism,

OpenMP was chosen as a simple shared memory parallelizing approach, which addresses most of todays workstations and shared memory computing cluster.

The remaining chapters of this thesis are divided into the following parts: Following this introduction will give a brief summary of the LBM model. Section 1.2 gives a numeric presentation of the Bouzidi boundary conditions and the mathematical base of the algorithm used for calculation. Chapter 2 then contains a small introduction to the *LBMKernel* and the implementation details of the *LBM2BC*. Together with the different test cases other features of the channel flow are presented. Finally an overview of the implementation of the unstructured list layout is given. Results are presented in chapter 3 for single processor performance and in chapter 4 results of the OpenMP simple parallelizing approach are shown. Finally chapter 5 summarizes the Thesis and gives a brief outlook.

1.1 The lattice Boltzmann method

The lattice Boltzmann method [7] [20] is a relative new technique to CFD. Classical CFD solvers are based on Navier-Stokes equations and thus solve large systems of non-linear partial differential equations in time t and space $\vec{\boldsymbol{x}}$. In contrast the LBM solves the lattice Boltzmann equation which is based on a simplified kinetic model. The physical background of the lattice Boltzmann method is given by the the Bhatnagar–Gross–Krook (BGK) [5] equation:

$$\partial_t f + \boldsymbol{\xi} \cdot \nabla f = -\frac{1}{\lambda}\left[f - f^{(0)}\right]. \tag{1.1}$$

$f^{(0)}$ is the Maxwell–Boltzmann distribution function, the distribution function $f(\vec{\boldsymbol{x}}, \boldsymbol{\xi}, t)$ is the basic quantity with the discrete velocity $\boldsymbol{\xi}$, and λ is the relaxation time. In a first step the continuous microscopic velocities are replaced by discrete velocities based on the chosen representation model. For 3D the two widely used models are D3Q15 and D3Q19, Qian et al. [15]. The notation of the DxQy models, defines x as the number of spatial dimensions and y counts the number of different discrete velocity directions. A representation of the D3Q19 model can be seen in figure 1.1. As one approximates the real motion of particles in any direction, with a finite amount of motion vectors, a more general representation would be the D3Q27 model, which however, does not significantly improve the stability and accuracy of the approximation. As a good trade off for stability and performance reasons ([13]) the D3Q19 model was chosen for this implementation. The finite set of velocities $\vec{\boldsymbol{e}}_{\boldsymbol{\alpha}}$ and the correspondent distributions $f_\alpha(\vec{\boldsymbol{x}}, t)$ descretize $\boldsymbol{\xi}$.

Inserted into the above formula 1.1 with $f_\alpha(\vec{\boldsymbol{x}}, t) = f(\vec{\boldsymbol{x}}, \boldsymbol{\xi}, t)$ and $f_\alpha^{(eq)}(\vec{\boldsymbol{x}}, t) = f^{(0)}(\vec{\boldsymbol{x}}, \boldsymbol{\xi}, t)$ as the equilibrium distribution function we get:

$$\partial_t f_\alpha + \vec{\boldsymbol{e}}_\alpha \cdot \nabla f_\alpha = -\frac{1}{\lambda}\left[f_\alpha - f_\alpha^{(eq)}\right]. \tag{1.2}$$

Considering that this method is applied to athermal fluids we get the equilibrium distribution function as stated in Qian et al. [15]:

$$f_\alpha^{(eq)} = w_\alpha \rho \left[1 + \frac{3}{c^2}\vec{\boldsymbol{e}}_\alpha \cdot \vec{\boldsymbol{u}} + \frac{9}{2c^4}\left(\vec{\boldsymbol{e}}_\alpha \cdot \vec{\boldsymbol{u}}\right)^2 - \frac{3}{2c^2}\left(\vec{\boldsymbol{u}} \cdot \vec{\boldsymbol{u}}\right)\right]. \tag{1.3}$$

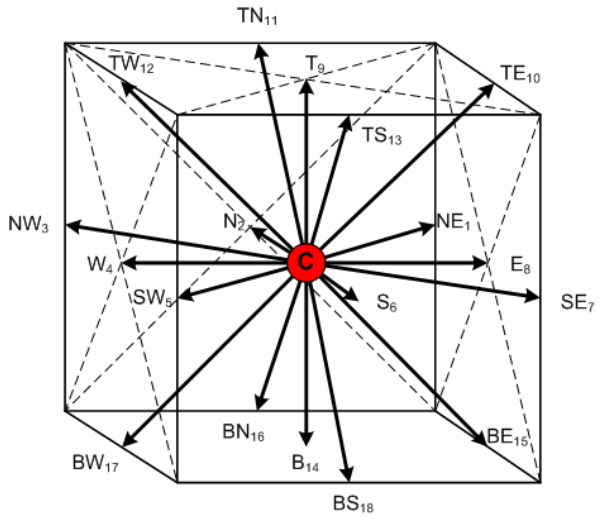


Figure 1.1: Discrete velocities in the D3Q19 model.

Here $\vec{\boldsymbol{u}}$ denotes the macroscopic velocity, ρ is the hydrodynamic density and $c = \frac{\delta x}{\delta t}$ represents the lattice speed, where δx is the lattice constant and δt the time step. The discrete set of velocities $\vec{\boldsymbol{e}}_\alpha$ is defined as follows:

$$\vec{\boldsymbol{e}}_\alpha = \begin{cases} (0,0,0), & \alpha = 0 \\ (\pm 1,0,0)\mathrm{c},(0,\pm 1,0)\mathrm{c},(0,0,\pm 1)\mathrm{c}, & \alpha = 2,4,6,8,9,14 \\ (\pm 1,\pm 1,0)\mathrm{c},(0,\pm 1,\pm 1)\mathrm{c},(\pm 1,\ 0\ ,\pm 1)\mathrm{c}, & \alpha = 1,3,5,7,10,11,12,13,15,16,17,18 \end{cases} \tag{1.4}$$

The weighting factor w_α for each discrete direction, is defined through:

$$w_\alpha = \begin{cases} \frac{1}{3}, & \alpha = 0 \\ \frac{1}{18}, & \alpha = 2,4,6,8,9,14 \\ \frac{1}{36}, & \alpha = 1,3,5,7,10,11,12,13,15,16,17,18 \end{cases} \tag{1.5}$$

Finally the hydrodynamic density ρ and momentum $\rho\vec{\boldsymbol{u}}$ of f and $f^{(eq)}$ can be calculated as follows:

$$\rho = \sum_\alpha f_\alpha = \sum_\alpha f_\alpha^{(eq)}, \tag{1.6}$$

$$\rho\vec{\boldsymbol{u}} = \sum_\alpha \vec{\boldsymbol{e}}_\alpha f_\alpha = \sum_\alpha \vec{\boldsymbol{e}}_\alpha f_\alpha^{(eq)}. \tag{1.7}$$

The pressure is obtained by the equation of state of an ideal gas $p = \rho c_s^2$ where $c_s = \frac{1}{\sqrt{3}}$ is the sound of speed of the model .
Discretizing equation 1.2 with finite differences upwind scheme with an explicit euler time step t and space $\vec{\boldsymbol{x}}$, with $|\delta_x| = |\delta_t|$ results in:

$$f_\alpha(\vec{\boldsymbol{x}}_i + \vec{\boldsymbol{e}}_\alpha \delta t, t + \delta t) = f_\alpha(\vec{\boldsymbol{x}}_i, t) - \frac{1}{\lambda}\left[f_\alpha(\vec{\boldsymbol{x}}_i, t) - f_\alpha^{(eq)}(\vec{\boldsymbol{x}}_i, t)\right]. \tag{1.8}$$

By comparing the results of this approach with the Navier Stokes equations, the relation for the kinematic viscosity ν can be derived theoretically:

$$\nu = \frac{1}{6}\left(\frac{2}{\lambda} - 1\right) \tag{1.9}$$

Finally equation 1.8 can be split into two steps:

$$\text{collision step:} \qquad \tilde{f}_\alpha(\vec{\boldsymbol{x}}_i, t) = f_\alpha(\vec{\boldsymbol{x}}_i, t) - \frac{1}{\lambda}\left[f_\alpha(\vec{\boldsymbol{x}}_i, t) - f_\alpha^{(eq)}(\vec{\boldsymbol{x}}_i, t)\right], \tag{1.10}$$

$$\text{streaming step:} \quad f_\alpha(\vec{\boldsymbol{x}}_i + \vec{\boldsymbol{e}}_\alpha \delta t, t + \delta t) = \tilde{f}_\alpha(\vec{\boldsymbol{x}}_i, t). \tag{1.11}$$

Where $\tilde{f}_\alpha$ denotes the distribution function after the collision step.
The algorithm determined by equations 1.10 and 1.11 is called collide–stream order or push method, as the distribution functions are read from the local cell. The updated values are streamed to the adjacent cells. Reversing these two steps is equivalent and yields the so called stream–collide order or pull method. As Iglberger [11] states the collide–stream order shows better performance, this version was chosen for close investigation. Besides the calculation of fluid cells the limiting obstacles, forming the channel borders, require a special boundary treatment. A simple approach for treating obstacle cells, the so called bounce-back, simply reflects all incoming, if we see the obstacle cell as the center, or all outgoing, if we see the fluid cell as the center, velocities back to its origin from the correspondent distribution function velocity. This method performs very well in terms of performance and ensured mass conservation exactly. However, the bounce-back implicitly assumes that the wall is located exactly at half way between fluid and obstacle node. Therefore, problems arising for complex geometries for example as curved surfaces or skewed walls and spheres, will be discussed in the next section and require another boundary treatment.

1.2 Bouzidi boundary condition of second order

The straightforward bounce-back approach results in a simple bounce back of the fluid/particles (reflexion) at the fluid to solid boundary. It is based on the following formula:

$$f_{\bar{\alpha}}(\mathbf{x}_f, t + \delta_t) = \tilde{f}_\alpha(\mathbf{x}_f, t), \tag{1.12}$$

where $f_{\bar{\alpha}}(\mathbf{x}_f, t + \delta_t)$ denotes the post-propagation distribution function, α is the current velocity or link direction. The correspondent link direction $\bar{\alpha}$ determines the link which stands towards the current link direction from the neighboring cell. For example links *8* and *4* are correspondent links in the D3Q19 model as defined in figure 1.1. Figure 1.2 shows the problem which arises with this simple approach. Every curved or simple inclined boundary, is reduced to a staircase approximation of the surface. The result is that this boundary condition loses accuracy in comparison to the natural geometry with increasing complexity, because it degenerates any curved geometry to a discrete grid spaced boundary. It degrades the second order accuracy of the scheme to one of first order ([12]). In order to avoid this degeneration Bouzidi's et.al. [6] proposed an advanced bounce-back with an interpolation scheme, as presented in figure 1.3. For the interpolation the actual distance ratio of fluid and obstacle between the link origin (C_a) and the destination cell (C_b) must be determined.

$$q = \frac{|\overline{C_a I}|}{|\overline{C_a C_b}|}. \tag{1.13}$$

Bouzidi's boundary condition with linear interpolation is obtained with the following equations (see figures 1.4 and 1.5):

$$f_{\bar{\alpha}}\left(C_a, t + \delta_t\right) = \tilde{f}_\alpha\left(H, t\right) = 2\,q\,\tilde{f}_\alpha\left(C_a, t\right) + (1 - 2\,q)\;\tilde{f}_\alpha\left(F_1, t\right) \quad , \qquad \text{for } q \leq \frac{1}{2}. \tag{1.14}$$

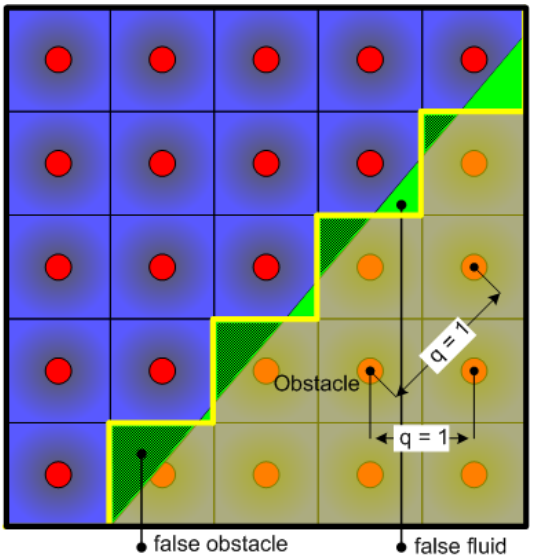


Figure 1.2: Discretization of standard bounce-back.

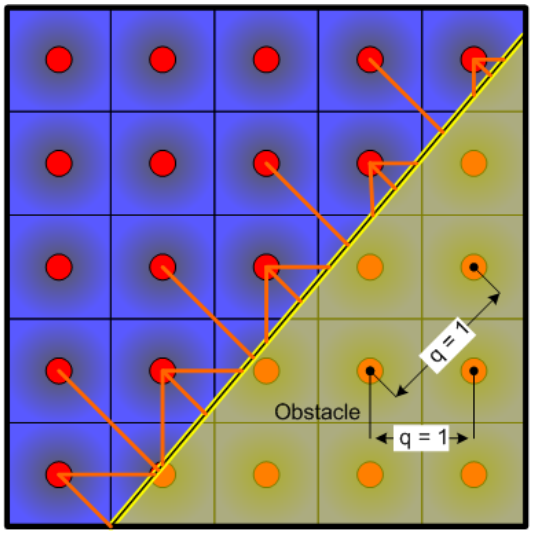


Figure 1.3: Schematic discretization of standard Bouzidi boundary condition.

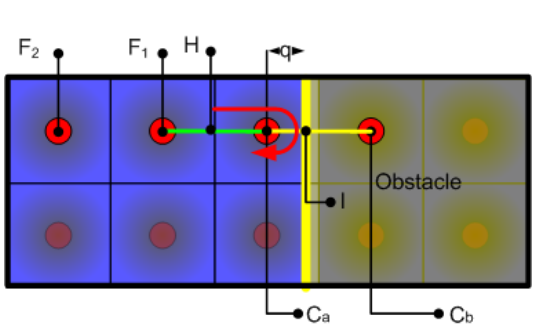
and

$$f_{\bar{\alpha}}\left(C_a, t+\delta_t\right) = \frac{1}{2\,q}\,\tilde{f}_{\alpha}\left(C_a, t\right) + \frac{2\,q-1}{2\,q}\,\tilde{f}_{\bar{\alpha}}\left(C_a, t\right) \quad , \qquad \text{for } q > \frac{1}{2}. \tag{1.15}$$

For $q \leq \frac{1}{2}$ equation 1.14 and figure 1.4 show that an virtual interpolation point H is introduced. As the distance between the cell center C_a and the boundary is less than unit lattice space, the velocities of the cell one layer behind also have to be taken into account. But this cell has a distance more than unit lattice space. Therefor the distance between point H and the boundary is chosen to be exactly unit lattice space. The velocities at point H are interpolated from the current cell C_a and the cell one layer behind F_1, following the correspondent link direction, and influence the resulting velocity depending on the q value correction as seen in formula 1.14. This involves the distance differences.
In contrast for values of $q > \frac{1}{2}$ equation 1.15 and figure 1.5 show that, the crossed link traverses the cell borders and so no interpolation point H is needed. The bounce-back velocities are all taken from the border fluid cell C_a and are corrected by the value of q. For $q = \frac{1}{2}$ equations 1.14, 1.15 are equivalent and they represent the simple bounce-back.

1.3 Ray plane intersection

For the inclined channel test cases the discrete velocity vectors from the D3Q19 model are checked for an intersection with the channel boundaries. The value of q is the fraction between fluid and obstacle part of one link. Therefor we need to calculate the intersection point between the boundary and the link. In 2D this was already done by Zschaek [21]. In 3D these equations are substituted by a ray and a plane, as shown in figure 1.6. The ray $\vec{\boldsymbol{R}}$



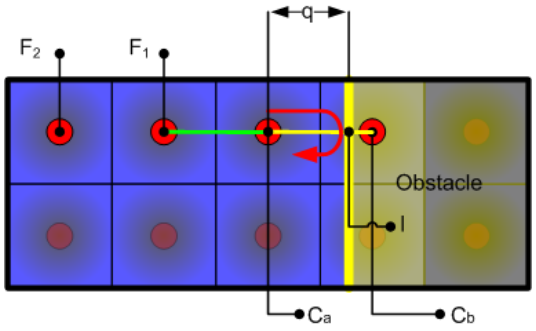


Figure 1.4: Bouzidi bounce-back for $q \leq \frac{1}{2}$. **Figure 1.5:** Bouzidi bounce-back for $q > \frac{1}{2}$.

is an parameterized vector with the origin in the center of the actual cell:

$$\begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + s \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \equiv \vec{\boldsymbol{R}}(s) = \vec{\boldsymbol{C}} + s * \vec{\boldsymbol{V}}. \tag{1.16}$$

Where $\vec{\boldsymbol{C}}$ is the center of the actual cell and $\vec{\boldsymbol{V}}$ is the vector to the neighboring cell, which can be obtained from the 19 discreet velocity vectors from equation 1.4. The plane is the lower or the upper skewed wall of the boundary and is defined by its normal vector $\vec{\boldsymbol{N}}, |\vec{\boldsymbol{N}}| = 1$ perpendicular to the surface and with lattice unit length, and the distance D between the origin and the plane:

$$\begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}, \quad D \qquad n_x * x + n_y * y + n_z * z + D = 0 \tag{1.17}$$

To determine if an intersection exists $\vec{\boldsymbol{R}}(s)$ is substituted into the plane equation yielding:

$$n_x(c_x + s * v_x) + n_y(c_y + s * v_y) + n_z(c_z + s * v_z) + D = 0 \Leftrightarrow \tag{1.18}$$

$$s_0 = -\frac{(n_x * c_x + n_y * c_y + n_z * c_z) + D}{n_x * v_x + n_y * v_y + n_z * v_z} \tag{1.19}$$

Inserting s_0 into the simple ray equation defines the intersection Point $\vec{\boldsymbol{I}}$.

$$\begin{pmatrix} i_x \\ i_y \\ i_z \end{pmatrix} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + s_0 \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{1.20}$$

Since $\vec{\boldsymbol{V}}$ was chosen from $\vec{\boldsymbol{e}}_{\boldsymbol{\alpha}}$ it is evident, that :

$$0 \leq s_0 = q = \frac{|\overline{C_a I}|}{|\overline{C_a C_b}|} \leq 1. \tag{1.21}$$

For a better understanding and right assignment of the different sections, figure 1.7 illustrates them in detail. One has to consider that straight links of the *D3Q19* model, like 2,4,6,8, have the unit length of one, but inclined links, like 1,3,5,7, have the length of $\sqrt{2}$. As the value of q always represents a ratio, there are the same values of q possible for different distances.

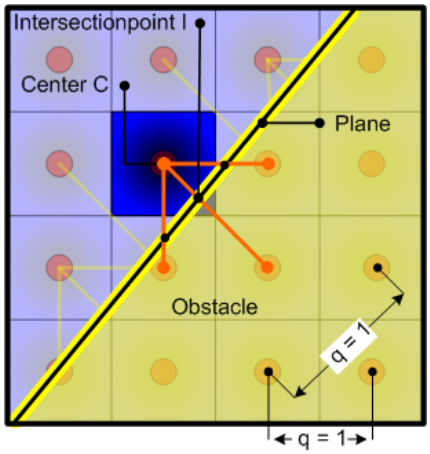


Figure 1.6: Intersection model of ray plane 2D segment.

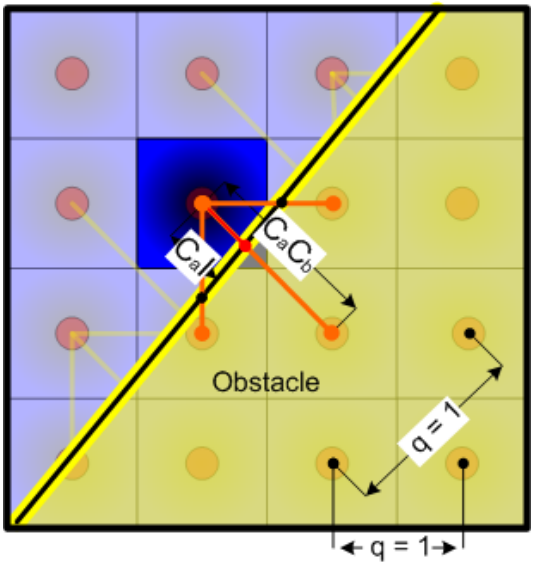


Figure 1.7: Intersection model of ray plane 2D segment with distances.

1.4 Ray sphere intersection

To apply the Bouzidi bounce-back with interpolation boundary condition to the inlying sphere too, an intersection test between the derived velocities and the sphere model must be performed. The definition of the ray remains unchanged, however $\vec{v}$ should be of unit length $|\vec{v}| = 1$, meaning $\vec{v} = \frac{\vec{e}_\alpha}{|\vec{e}_\alpha|}$, which still originates from the 19 discreet velocity vectors from equation 1.4:

$$\begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + s \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \equiv \vec{R}(s) = \vec{C} + s * \vec{v} \tag{1.22}$$

A sphere is basically defined by its center point T and a sphere radius r:

$$\begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \quad , \quad r \qquad \text{where} \qquad (x - t_x)^2 + (y - t_y)^2 + (z - t_z)^2 = r^2 \tag{1.23}$$

Taking the correspondent ray equation and substituting for x,y,z we get the following equation:

$$(c_x + s * v_x - t_x)^2 + (c_y + s * v_y - t_y)^2 + (c_z + s * v_z - t_z)^2 = r^2 \tag{1.24}$$

As this is a quadratic equation of the form :

$$A * s^2 + B * s + C = 0 \tag{1.25}$$

A,B,C could be defined:

$$A = v_x^2 + v_y^2 + v_z^2 = 1, |\vec{v}| = 1 \tag{1.26}$$

$$B = 2 * (v_x * (c_x - t_x) + v_y * (c_y - t_y) + v_z * (c_z - t_z)) \tag{1.27}$$

$$C = (c_x - t_x)^2 + (c_y - t_y)^2 + (c_z - t_z)^2 - r^2 \tag{1.28}$$

This yields the quadratic solution:

$$s_0, s_1 = \frac{\left(-B \pm \sqrt{(B^2 - 4 * C)}\right)}{2} \tag{1.29}$$

The calculation of the discriminant is of interest if the result is > 0, as the case < 0 indicates no intersection and case $= 0$ determines a tangent. With an positive discriminant, still two results are available, because the ray enters the sphere on the one side and leaves on the other. Inserting $s_m = min\,(s_0, s_1)$ into the ray equation,

$$\begin{pmatrix} i_x \\ i_y \\ i_z \end{pmatrix} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + s_m \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}, \tag{1.30}$$

gives the intersection point $\vec{I}$, thus the distance from the fluid cell center to the intersection point $\overline{C_aI}$ and the distance of the two cell centers $\overline{C_aC_b}$, also shown in figure 1.8. Which leads to the definition of the value of q for the sphere intersection:

$$0 \leq q = \frac{|\overline{C_aI}|}{|\overline{C_aC_b}|} \leq 1. \tag{1.31}$$

The q value must be inside the interval from zero to one, because any other value would not be between the two cell centers.

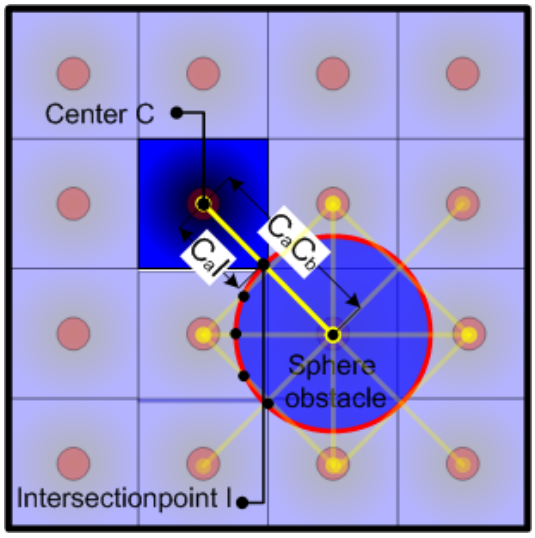


Figure 1.8: Intersection model of ray sphere 2D segment with distances.

Chapter 2

Basic implementation

The past chapter introduced the numerical terms and principles of the lattice Boltzmann method. It showed that a more challenging geometry, skewed walls or curved boundaries, overstrains the simple bounce-back scheme. Due to the loss of accuracy we introduced a new bounce-back with interpolation which provides a higher accuracy, at the cost of additional computational effort. To understand the performance impact, arising from this improved numerical method, knowledge about its implementation is indispensable.
To retain the sequence as in the chapter before, section 2.1 gives you an introduction to the standard *LBMKernel* as described by Donath [8]. We provide a brief introduction to the current version used, whereas the emphasis is on the basic algorithm structure, to produce a common basis for more advanced topics and on the improved segments of the code, which are not covered by past publications. Section 2.2 will show the implementation specific differences between the simple bounce-back and the bounce-back with interpolation. After that the test cases used in this thesis are described. Finally this chapter shows the implementation of the unstructured list layout in 2.4.

2.1 Lattice Boltzmann implementation

As a starting point the fastest data layout, developed by Donath [8] was chosen, to investigate the more complex boundary conditions on contemporary architectures. Unlike many straight forward implementations the data layout represented by a 5 dimensional array, was changed to {*tNow/tNext*,*19 velocities*,*i-dimension*,*j-dimension*,*k-dimension*} in order to have each velocity for all cells consecutive in memory as described in [19], thus as the slowest running index. Although this data layout has some performance penalty considering only read operations from the grid, the performance improves significantly when writing and reading is considered together. The standard implementation traversed the lexicographical grid and calculated the complete collision and propagation for each cell after the other. In order to improve performance the innermost loop of the update process was split up into 3 parts which are carried out respectively for all cells. The first loop only calculates the macroscopic quantities from the current cell. The next two loops calculate the relaxation and stream the up to date values to the surrounding cells, first from cell center value to link number eight, then from link nine to link number 18. The standard lexicographic two grid layout, eliminates data dependencies and allows a custom traversing of the domain.

Besides the actual computation a preprocessing step is required. In the preprocessing phase the two grids, distinguished by the timestep variable, are allocated, as well as one obstacle

field. This field determines whether a cell is fluid or obstacle. Additionally there is a list with the coordinates of all boundary cells. In the actual calculation step, first the collision function is called and afterwards the bounce-back step is performed. As described in section 1.1 the equations could be split up into a collision equation 1.10 and an propagation equation 1.11. Not exactly following this theory, the implementation splits up the collision and propagation phase in a different way (Algorithm 2.1), as described before.

Algorithm 2.1: Collide-stream step of LBM

```
1   do k=1,kEnd
2
3       do j=j , jEnd
4          do i=1,iEnd
5   !if fluid
6   ! Read all particle velocities from current cell
7
8   ! Store all macroscopic  values in temporal arrays
9   !endif
10
11         enddo
12         do i=1,iEnd
13  !if fluid then
14  !Collision and propagation for velocities 0,1,2,3,4,5,6,7,8
15  !endif
16         enddo
17
18         do i=1,iEnd
19  !if fluid then
20  !Collision and propagation for velocities 9,10,11,12,13,14,15,16,17,18
21  !endif
22         enddo
23      enddo
24  enddo
```

The bounce-back routine traverses now the list with all obstacle cells (Algorithm 2.2). The obstacle cells were determined in the preprocessing phase and the coordinates were written into an index array called *barrierIdx*. The collision routine propagates velocities to other fluid cells, as well as to the adjacent obstacle cells this is shown in figure 2.1. These values are now reversed following the law of reflexion and written back to a fluid cell, as shown in figures 2.2 and 2.3. With little effort any geometry, limited by the grid itself though, can be modeled. However the accuracy lacks due to the fixed grid discretization described in section 1.2, which is not intercepted by the simple bounce-back.

2.2 Bouzidi bounce-back implementation for complex geometries

As Bouzidi's et. al. [6] boundary conditions take the real distance between fluid cell center and obstacle boundary into account, there are additional preprocessing steps to be done before the actual calculation. The main test case for this thesis was a skewed channel with different inclinations. Thus, as a first step a mathematic model of the boundary planes and sphere, as described in section 1.3 and 1.4, is needed. At each grid position the value of the two planes is calculated and compared to the actual fluid cell center. This can be done in a similar way for the sphere model. If a cell center lies between the two boundary planes,

Algorithm 2.2: Bounce-back of *LBMKernel*

```
1
2
3  subroutine bounceback(pdf,tNow,barrierIdx,barrierCnt,iEnd,jEnd,kEnd,obstacleField)
4
5
6    do m=1,barrierCnt
7
8  !Getting obstacle cell positions
9        i=barrierIdx(1,m)
10       j=barrierIdx(2,m)
11       k=barrierIdx(3,m)
12
13 ! Performing simple bounce back
14
15   enddo
16
17 end subroutine
```

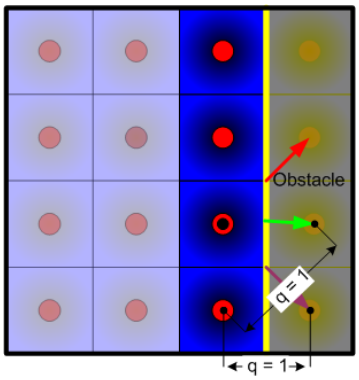


Figure 2.1: Streaming into obstacle cells.

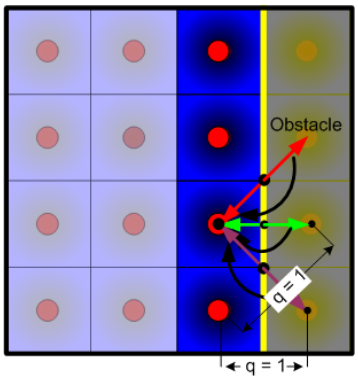


Figure 2.2: Actual "bounce-back".

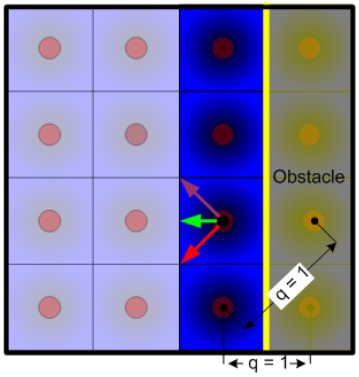


Figure 2.3: Streaming in reverse direction.

but outside the sphere, it is considered as a fluid cell in an array named *obstacleField*, which contains the type of a cell for all grid coordinates. The next step is to evaluate which links are cut by the boundary. To determine this, each fluid cell having a link with an obstacle is checked in the following way. For all links of this cell which have a correspondent link in an obstacle cell, the intersection point as described in 1.3 and 1.4 is calculated. If the intersection point lies on the link between fluid cell center and obstacle cell center, the value of q is calculated. The position of the actual fluid cell, the link number and the value of q are stored in four separate arrays. Q values greater than 0.5 are stored in *bc2IdxhighQ* and values of q less than 0.5 are stored in *bc2IdxlowQ*. The corresponding coordinates and link numbers are stored in *bc2Idxhigh* and accordingly *bc2Idxlow*. At the actual calculation the collision routine stays unchanged. Following Bouzidi's research, two different algorithms are applied to the high and low values of q. As these values are separated in different lists an easy traversing without further branches is possible. This enhances the *LBMKernel* with the Bouzidi boundary conditions (BBC) which leads to a new kernel called *LBM2BC*.

Algorithm 2.3: Bouzidi boundary conditions

```
!------------------------------------------------------------------------------
! Bouzidi calculations of high q-value link
!------------------------------------------------------------------------------


do m=1,qhighCnt

    i=bc2Idxhigh(m,1)
    j=bc2Idxhigh(m,2)
    k=bc2Idxhigh(m,3)

    pdf(i,j,k,offsetField(1,bc2Idxhigh(m,4)),tNext)    = &
          1/(2 *  bc2IdxhighQ(m) )* &
          pdf(
          i, &
          j, &
          k, &
          bc2Idxhigh(m,4), &
          tNext) + &
          (1 -( 1 /( 2 *  bc2IdxhighQ(m)))) *&
          pdf
          (i ,&
           j ,&
           k ,&
           offsetField(1,bc2Idxhigh(m,4)),&
           tNext)

enddo

!------------------------------------------------------------------------------
! Bouzidi calculations of low q-value links
!------------------------------------------------------------------------------

do m=1,qlowCnt

    i=bc2Idxlow(m,1)
    j=bc2Idxlow(m,2)
    k=bc2Idxlow(m,3)

    pdf(i,j,k,offsetField(1,bc2Idxlow(m,4)),tNext)     = &
          ( 2 *   bc2IdxlowQ(m) *&
          pdf&
          (i,&
           j,&
           k,&
           bc2Idxlow(m,4)),&
           tNext)  )+&
          (1- (2 *  bc2IdxlowQ(m) )) *&
          pdf&
         (i- offsetField(2,offsetField(1,bc2Idxlow(m,4))),&
          j- offsetField(3,offsetField(1,bc2Idxlow(m,4))),&
          k- offsetField(4,offsetField(1,bc2Idxlow(m,4))),&
          bc2Idxlow(m,4),&
          tNext)

enddo
```

2.3 Test cases

To discuss the impact on performance of the different boundary conditions both LBM algorithms were compared on open channel layouts without any obstacles. The domain was increased, starting at 16 x 16 x 16, over several iterations to finally 141 x 141 x 141. An example with a domain size of 64 x 64 x 64 can be seen in figure 2.5. The Bouzidi et.al. [6] boundary conditions are only applied to the bottom and top of the skewed channel and to the sphere surface. To accelerate the fluid, one segment of fluid cells in the center of the channel was continuously forced to move, by increasing the density on one side of the acceleration segment while decreasing the density on the opposite. The pressure decrease before these acceleration cells can be seen in figure 2.4, as generated by the CFD visualization software Tecplot [18].

In contrast to a given velocity at the inlet and a pressure outlet boundary condition, this kind of forcing needs periodic boundary conditions to generate the continuous flow needed. For this reason all outflow values from outflow (inlet) were reinserted at the inlet (outflow). Additionally periodic boundary conditions were introduced for the front-/backside too.

Moving to more complex surfaces, the first step is to skew the channel. The channel flow opening, thus the maximum count of fluid layers in the dimension k, is determined by the variable *FlowOpening*. In figure 2.5 a fully open channel is shown, where the *FlowOpening* is set to the maximum expanse of the k dimension, called *kEnd*. The shown macroscopic quantity is the fluid speed (sum of $|u_x, u_y, u_z|$), whereas red marks the highest speed and blue no motion at all. With an reduced *FlowOpening*, as shown for $FlowOpening = 32$ in figure 2.6, the amount of fluid cells decrease while the amount of obstacle cells at the boundary planes grows. To further increase the bounce-back treated cells, a sphere model is placed inside the channel. These inclined channel layouts are presented with an common domain of 64 x 64 x 64, $FlowOpening = 48$ and a sphere with $SphereRadius = 8$. The sphere is placed with contact of the outer boundary in 2.7 (i-coordinate = 5)and then shifts inside the channel as shown in figures 2.9 and 2.11. Figures 2.8, 2.10 and 2.12 show the same from the opposite view onto the channel.

In another test case with the domain size 64 x 64 x 64, the channel is reduced beginning at the fully open channel to a minimum of half the channel height, which can be seen in figures 2.5 and 2.6. This has an impact on the overall fluid cell count and onto the count of cells which need a boundary treatment.

The amount of fluid cells inside the channel (FC) of a d^3 domain is estimated with:

$$FC = FlowOpening * d * \sqrt{d^2 + (d - FlowOpening)^2}, \tag{2.1}$$

and the amount of total cells (TC):

$$TC = d^3 \tag{2.2}$$

This leads to the fluid to total cell ratio:

$$\frac{FC}{TC} \approx \frac{FlowOpening}{d}. \tag{2.3}$$

Yielding, that with a decreased *FlowOpening* the number of treated fluid cells decrease too, while the amount of memory usage stays constant and the amount of boundary cells (BC) rises:

$$BC = 2 * d * \sqrt{d^2 + (d - FlowOpening)^2}. \tag{2.4}$$

As you can see in figure 2.13 the performance drops with the decreasing *FlowOpening*. That led to the implementation of a new data layout, called unstructured list, which should be insensitive in terms of obstacle presence inside the channel as stated by Donath et.al. [10].

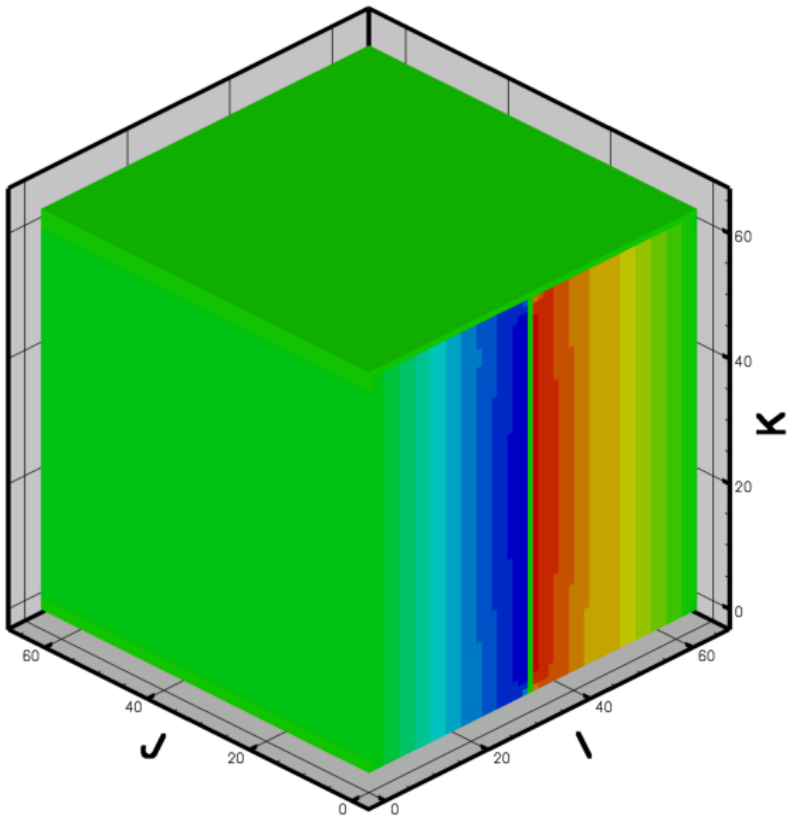


Figure 2.4: *64^3 domain with FlowOpening* $= 64$.

Pressure values are ranging from high pressure, marked with orange to low pressure with blue. Flow is going from left to right. The top and the bottom are fixed obstacles.

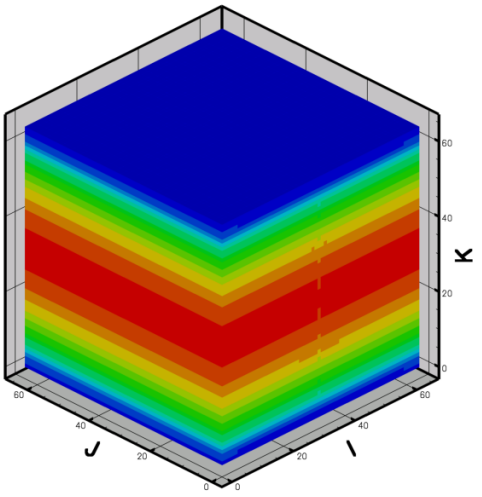


Figure 2.5: Visualization of speed. 64^3 domain with *FlowOpening* = *64*.

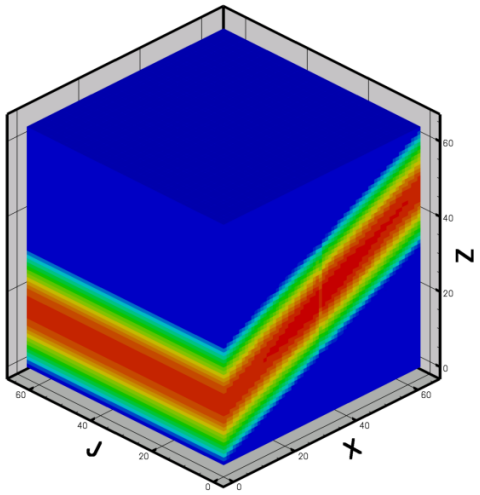


Figure 2.6: Visualization of speed. 64^3 domain with *FlowOpening* = *32*.

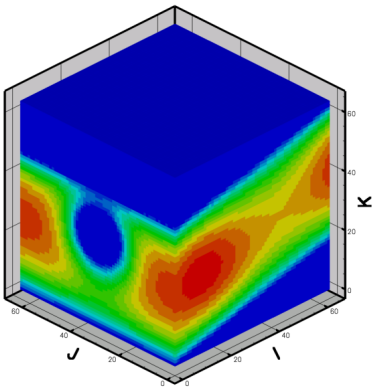


Figure 2.7: Visualization of speed. 64^3 domain with sphere distance to wall = 8 front view.

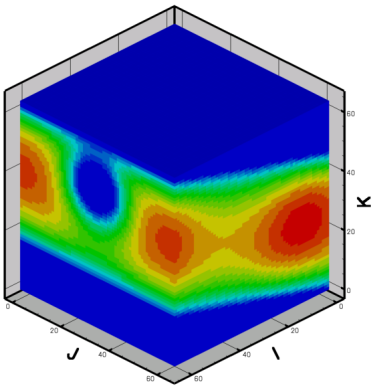


Figure 2.8: Visualization of speed. 64^3 domain with sphere distance to wall = 8 back view.

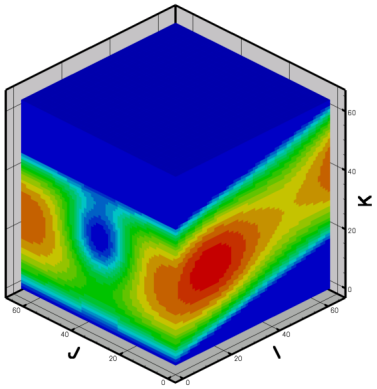


Figure 2.9: Visualization of speed. 64^3 domain with sphere distance to wall = 12 front view.

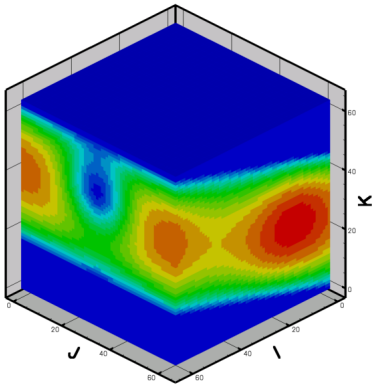


Figure 2.10: Visualization of speed. 64^3 domain with sphere distance to wall = 12 back view.

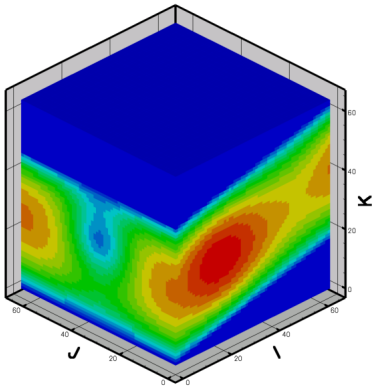


Figure 2.11: Visualization of speed. 64^3 domain with sphere distance to wall = 16 front view.

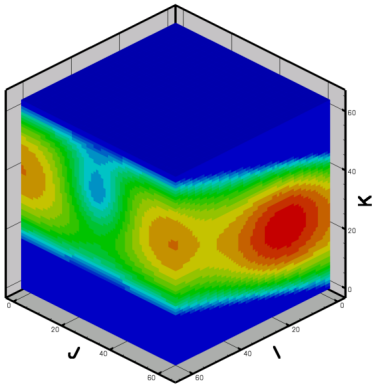


Figure 2.12: Visualization of speed. 64^3 domain with sphere distance to wall = 16 back view.

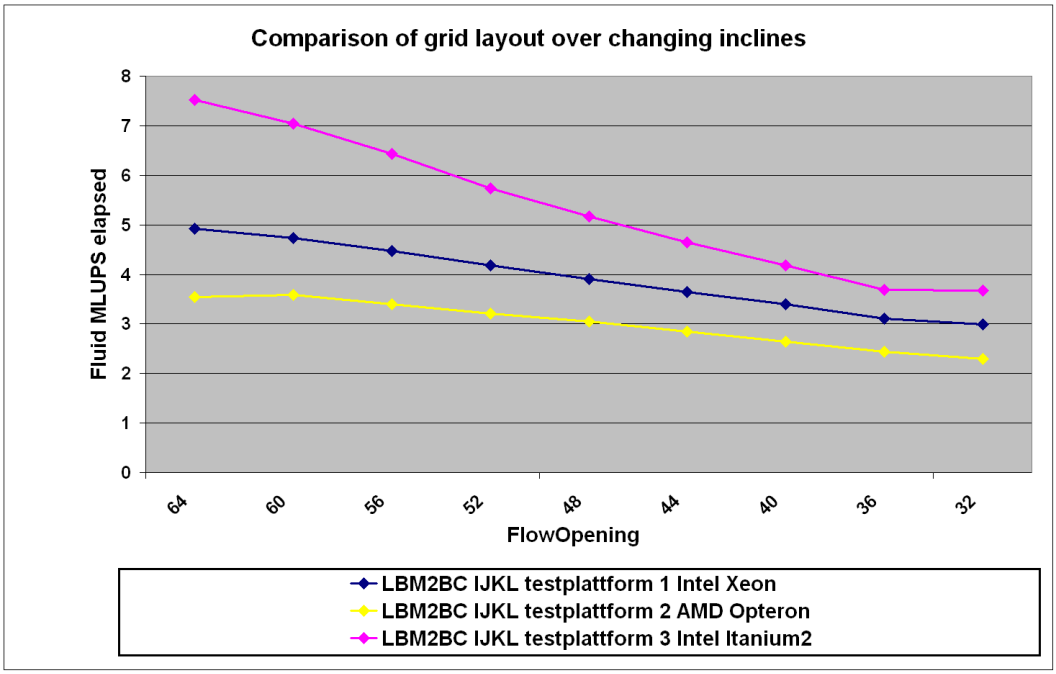


Figure 2.13: Comparison of fluid MLUPS on different test systems. For a 64^3 domain, the FlowOpening starts with an full opened channel ($FlowOpening = 64$) and is decreased to the half of this value.

2.4 Unstructured list layout for complex geometries

2.4.1 Motivation

The lexicographic grid layout is the most common domain representation used, as this model maintains the original shape of the domain with respect to the limits of the discretization resolution. Especially for the LBM, this representation is very suitable, because the neighbor cells can be easily accessed through simple offset operations, which stay the same for each link over the whole grid. Considering the standard problem, a simple channel with boundaries on the outside with no or small incline (figure 2.14), the lexicographic grid representation is the favorite. However in case of a skewed channel with increasing incline, the amount of obstacles which have to be stored in the lower and upper triangular besides the skewed channel as seen in figure 2.15 increases.
As a consequence the obstacle cells with no link to any fluid cell (fixed obstacle), are in fact neither calculated nor touched in any manner, storing these elements means wasting memory. Modern compilers and processors use prefetching to get data according to the program access pattern. If the pattern is not too difficult or unpredictable, the performance improvements can be tremendous. In the case of a skewed channel, such prefetching mechanism however, would load dispensable fixed obstacle data on a regular basis, i.e. would waste the rare resource memory bus.

Finally certain domain sizes could not be simulated, although the count of fluid cells should be below the allocatable memory space. Obviously the solution is not to store the fixed obstacles and only have the fluid cells and the obstacles at the boundary. A potential approach to this

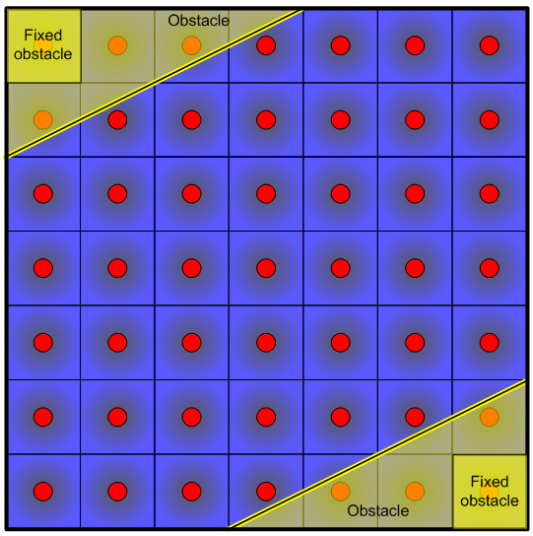


Figure 2.14: Unused obstacle cells (fixed obstacle) in case of small incline.

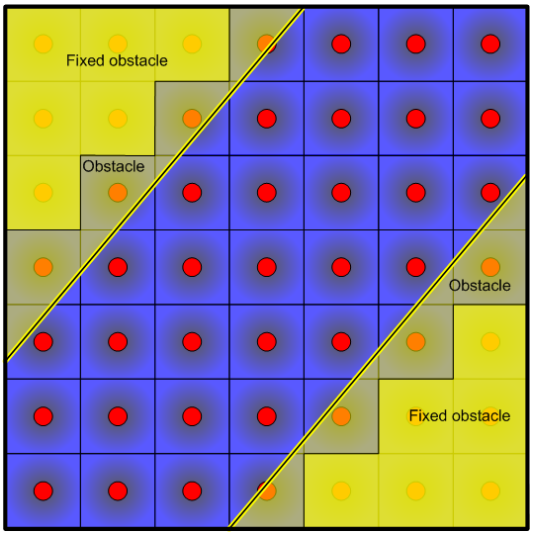


Figure 2.15: Unused obstacle cells (fixed obstacle) in case of great incline.

problem is evaluated with the unstructured grid in the following section.

2.4.2 Implementation

The unstructured grid is actual an unstructured list called *pdfList*, in what follows that velocities of all fluid related cells (fluid cells and neighboring obstacles) are stored there. At the initialization step three loops, traversing a standard 3D grid fill this list and generate a standard LIJK layout, thus all velocities of one cell lie consecutive in memory. The meaning of unstructured becomes obvious when the collision step is tried to be performed. First the velocities of the current cell are needed. As these lie consecutive in memory, a base counter always points to the start of a new cell. To propagate these values to the neighboring cells, the destination index inside the list must be known, as shown in algorithm 2.4. Therefor an additional *correspondentlist* is necessary. This list has as many entries as the *pdfList* and points for each velocity to the cells equivalent link neighbor. Now for each link which should be propagated, the correspondent neighbor is acquainted and can be used as seen in algorithm 2.4. To build this list, the lexicographical offset operations mentioned in section 2.1 are used.

At initialization time another 3D grid called *pdfLookup* is generated. For each grid point copied to *pdfList* the starting list index is stored to its coordinates in *pdfLookup*. That way a somewhat map, of the list is produced as seen in the flowchart 2.17. Again memory is allocated for the hard obstacle cells, but now only one integer (platform dependent 4 to 8 Byte) and not 19 double values (8 Byte). The setup of the *correspondentlist* utilizes the simple 3D offset operations to gather the neighboring cells list indices and stores them to the *correspondentlist*. These operations are illustrated in flowchart 2.17. The collision algorithm works independently from *pdfLookup* on the *pdfList* and *correspondentlist*. The Bouzidi et al. [6] boundary condition is treated similar to section 2.2. Again four lists contain the required information, but instead of storing the coordinates, the list indices of the cells treated

are stored. As the *pdfList* contains all values of interest, which must be calculated, only one single loop traverses the list and performs the collision. Inside algorithm 2.4 *tNext* defines the next timestep offset and *ImOmega* and *ne0,ne1* are viscosity and discreet velocity values necessary for computation. Of course the code snippet does not present a full implementation and is only for illustration. The treatment of the periodic boundary conditions and forcing routines utilize the *pdfLookup*, but are irrelevant in terms of performance.

Algorithm 2.4: List based collision

```
1  subroutine relax_list
2
3    !=========================================
4    ! Collision relax...
5    !=========================================
6
7  int (I4B) ::currentCell ,currCell
8
9    do currCell=0,pdf_listCount
10
11     ! Collision step: calculating macroscopic quantities of current cell( density ,
           velocity )
12
13     currentCell = currCell * 19
14
15     !Example propagation of the first two velocities
16      pdf_list(correspondent_list(currentCell + 1)    + tNext ) =   &
17               pdf_list(currentCell + 1     tNow) * ImOmega + ne0 )
18
19      pdf_list(correspondent_list(currentCell + 1 + 1)   + tNext) =   &
20               pdf_list(currentCell + 1 + 1  +  tNow )     * ImOmega + ne1
21
22     ! And so on for the following 16 links
23
24   enddo
25
26 end subroutine relax_list
```

2.4.3 Optimization of unstructured list layout

A great disadvantage of general indirect addressing is the hidden data structure. An unstructured list layout has more management effort and less chances on compiler optimizations, because of the non-constant scattered data-accesses. As a consequence a popular technique of enhancing spatial data locality, called blocking was applied. In the case of the lattice Boltzmann method, the traversing of the grid is altered to a traversing of a block, which ranges over three dimensions. As seen in figure 2.18 for 2D, the elements 1 to 9 of block 1 are consecutively traversed. For 3D the behind lying cells are traversed as well. For a cell update of cell one we need the velocities of cell 1 and cell 2,4 and 5 for the propagation. After cell 1 is finished, the update for cell 2 needs values from cell 1,3,4,5 and 6 inside the block. Comparing these two collision steps, the values of cell 1,2,6 are used in both steps. For the cell 3 update, values of cell 2,3,5 and 6 are needed. Again these cells coincide with the before used cells. Although values needed from outside the block were not mentioned, this technique still enhances the cache reuse. After finishing one whole blocked area, the same technique is applied to the next area, thus the block is pushed through all dimensions of the domain. The schematics only illustrate a 2D domain, but the technique is simply extended

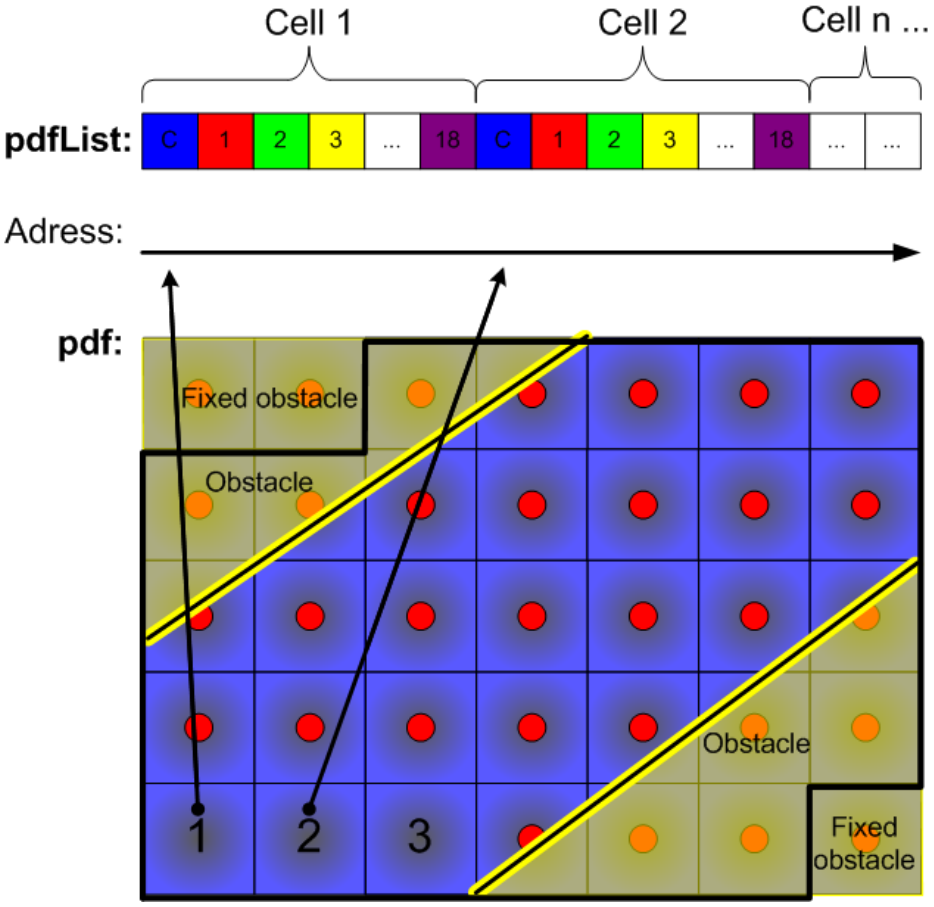


Figure 2.16: Relation between *pdf* and *pdfList*.

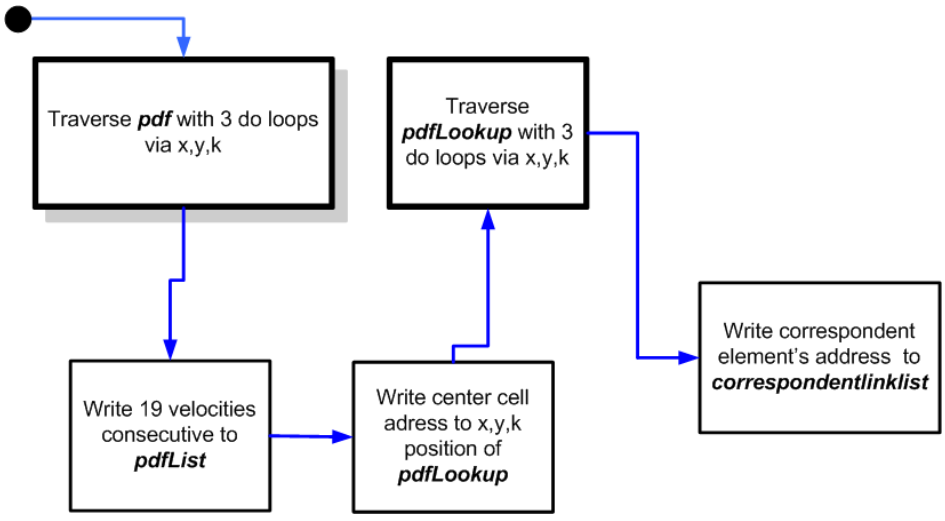


Figure 2.17: Flow chart of preprocessing list phase.

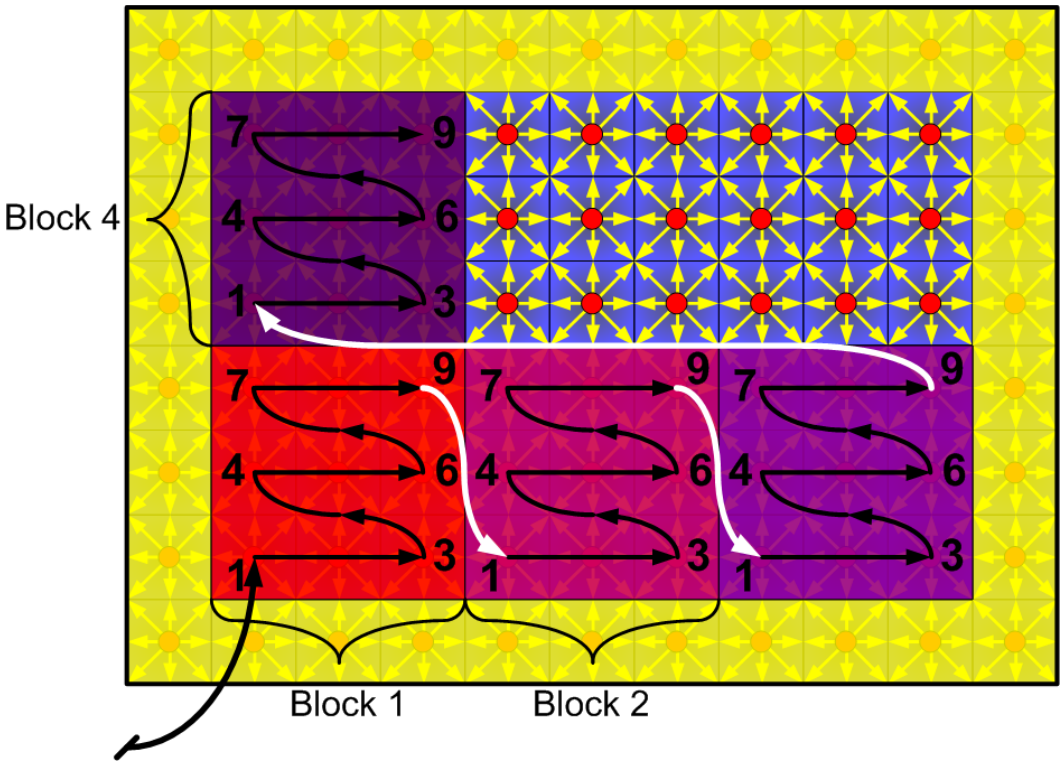


Figure 2.18: Traversing of blocked areas.

to apply blocking on three dimensional domains. For the purpose of better visualization, a sample blocksize of 3 was chosen.

In Iglberger [11] and Donath [8] the 3D blocking was applied inside the collision routine of an grid bases LBM algorithm. This has the disadvantage, that three loops, one for each dimension, where supplemented by additional three loops. The outer loops then traversed the dimensions with a stride regarding the block size, while the inner three loops traverse the blocks inside the stride. Overall the loop overhead increased, because of the small iteration count on the innermost loops. As the list collision only traverses the 1D list space, there is no reasonable approach to apply the blocking here. However with altering the initial setup of the *pdfList* the blocking can be applied too. Therefor the above mentioned three additional do loops are implemented into the setup routine and collision routine traverses still a 1D list but the way through the list has been optimized for a maximum cache reuse in the setup phase already. Furthermore, any change in layout only has to be done inside the initialization routine. This removes the negative side effects of additional loop overhead, as the collision routine maintains the original long running loop. The blocking management work is only done once inside the preprocessing phase.

Iglberger [11] states that an blocksize of 8 x 8 x 8 should bring most possible improvement regarding 3D spatial blocking, because it is the largest cubic blockingsize, within the powers of 2, that fits into a cache of 512 KB or 1 MB. From the estimation that a d^3 block (BMU) needs:

$$BMU = d^3 * 19 * 8 * 2 \text{ Bytes} \tag{2.5}$$

Where first the number of cells, followed by the number of links and then by the Bytes per element and the two time steps are shown. A blocking factor of eight would lead to a used cache size of $152KB$ and a blocking factor of 16 already exceeds 1 MB of cache with 1216 KB of cache usage.

Taking the LBM memory access pattern into account, cells from the upper, lower and the same layer as the current cell are needed for a cell update. Whereas up and down refers to traverse the k-dimension. Though all cells more than two layers distant from the current k-layer have no benefit of the blocking scheme. Hence, an optimal blocking factor extends to a three layer 2D block, which is then pushed through the domain. This leads to different blocking factors for different cache sizes, based on the formula:

$$BMU = d^2 * 3 * 19 * 8 * 2 \text{ Bytes.} \tag{2.6}$$

Where first the 3 layers, followed by the 19 velocity values with 8 Bytes each and the timestep is shown. Additional the correspondentlist has an estimated memory usage of:

$$CMU = d^3 * 19 * 4 * 2 \text{ Bytes.} \tag{2.7}$$

Considering that the maximum amount of cache is not available to the blocked data, because of other data, cache lines and common cache associativity problems. Hence, an optimal blocking factor is less than the maximum possible block size that fits into the cache. Detailed parameter studies of the evaluated test systems are presented in chapter 3, which provide the optimal blocking factor for each platform.

Chapter 3

Single processor performance

In the previous chapters an introduction to the numerical basics of the fluid dynamic of the LBM method was made. A brief implementation overview of the original *LBMKernel* was given. The implementation specific details of the enhanced *LBM2BC* were shown and the development of the unstructured list layout was explained. The *LBM2BC* algorithm is no longer working on a domain with fluid as the main share. Till now, the basic layout characteristics were the three space coordinates. With the *LBM2BC* the fluid cell count and the obstacle cell count can change over one input parameter, the incline. Although the space coordinates stay constant, this impact has to be considered. This leads to the questions if the enhanced *LBM2BC* can stay close to the performance achievements of the *LBMKernel* or whether the list layout has benefits or is dedicated to even more complex and disturbed geometries.

To measure performance we count the lattice site updates which were performed per second. In general this would be the domain size times the iterations and divided by the elapsed time in seconds. The inclined channel however with its sphere inside counts the amount of fluid cells separately which replaces the domain size and leads to the real fluid performance. This leads to the scale unit fluid MLUPS. Where fluid determines that performance regards only the real fluid cells and the abbreviation MLUPS means one million lattice site updates per second.

3.1 Test systems

For the performance evaluation three widespread platforms were chosen. The most common platforms available today are IA32 based computing nodes which are represented by an Intel Xeon [3] based dual node. Although an AMD Opteron is a IA32 based processor with AMD64 too, it has a more sophisticated memory architecture with high bandwidths and low latencies. Thus another test system based on a quad node AMD Opteron 848 [17] [16] was evaluated as well. To represent the high end of non-vector processors an Intel Itanium2 based dual node was chosen. An overview of these systems specifications is given in table 3.1 and 3.2.

The key features of the Intel Xeon processor are the high clock rate and the long, up to 30 stage (Netburst), pipeline. It is capable to do one fused multiply add operation (SIMD) at one clock cycle and has the out-of-order execution ability. This means that the processor hardware unit may reschedule instructions dynamically, to improve performance. On the dual nodes, two processors share the same memory bus which is organized by a centralized

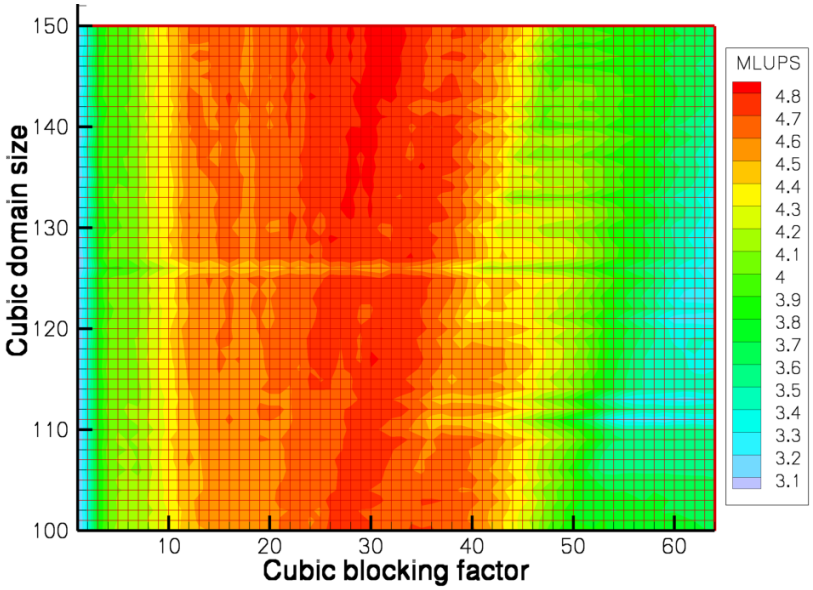


Figure 3.1: Parameter study of blocking factors for the unstructered *LBM2BC* list layout LIJK on test system 1 / Intel Xeon.

memory management unit. As a consequence a non memory intensive algorithm could speed up by a factor of two, when using the second processor. However for memory intensive codes, even a slowdown is possible, because of the increased overhead when accessing the memory through the shared bus. This effect will be seen with the OpenMP variant. According to the cache size of test system 1 / Intel Xeon, and the estimations done in equation 2.5, 16 should be the best performing blocksize. Regarding figure 3.4 an optimal blocking factor would be 38. Based on equation 2.6 and the real available cache space, an optimal blocking factor would range between 26 and 40. The improved blocking factor yields to be 28, which is in very agreement with the measurements presented in figure 3.1.

The more sophisticated memory architecture of the AMD Opteron processors is better adapted to these needs. First each processor has its own on-chip memory controller, which accesses the local memory. Depending to the model there are 2 to 4 so called HyperTransport links, each connecting the processor to the other processors. On a 4-way system each CPU is connected to 2 neighbor processors as seen in figure 3.2. To access the memory of the fourth processor, in the opposite corner, 2 HyperTransport links have to be used. As this remote access has a higher latency, the Opteron platform is a ccNUMA platform, were additionally cache coherence protocols guarantee valid data in each processors cache. This allows the AMD Opteron based systems to manage a large amount of memory and to scale the available memory bandwidth. For both IA32 platforms the Intel Fortran EM64T 9.0_021 Build 20050430 was used [4]. The commonly used flags were *-O3*, *-tpp7*, *-xP* for Intel Xeon , *-xW* for AMD Opteron and *-openmp* for OpenMP.

The Itanium2 is a new processor architecture with a new execution model approach in contrast to existing processors. Unlike to the out-of-order execution, which has it advantages in fast branching and altering calculation patterns, the Itanium2 [2] processors follows the EPIC (explicit parallel instruction computing) model. This model needs a compiler that generates

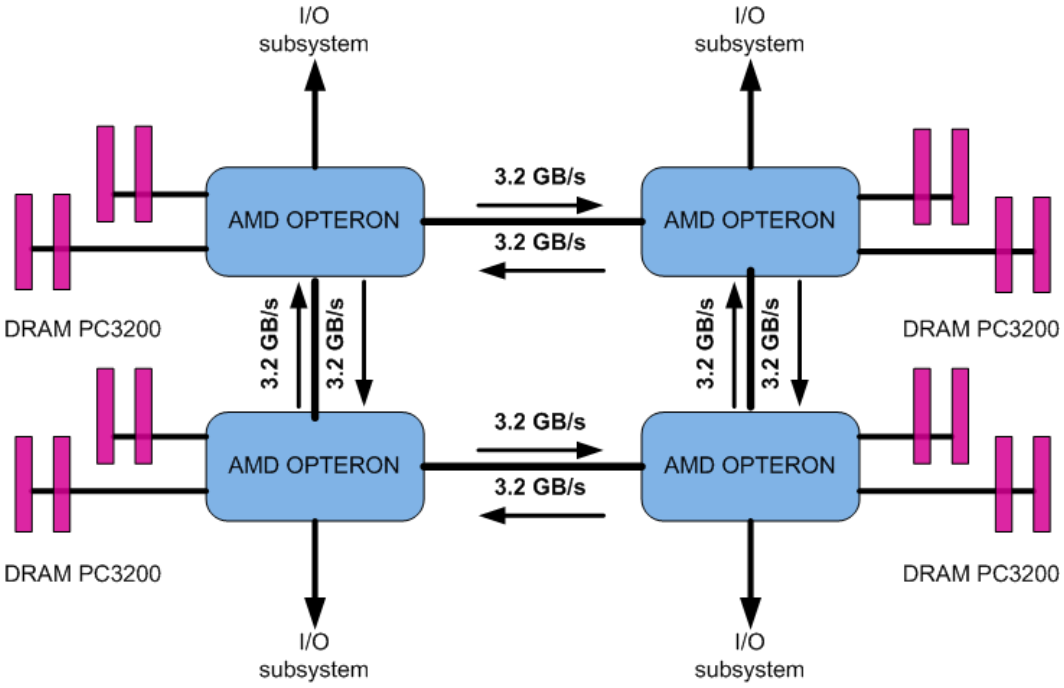


Figure 3.2: Schematic AMD Opteron quad node interconnect.

large streams of independent instructions. The processor follows these executions with no exceptions. All unpredictable code branches cause a loss of performance. An exception are the *if* branches which are covered by the highly efficient predicate registers. Like vector processors the Intel Itanium2 performs best on long running loops. The memory subsystem is also based on a bus controller, like the Xeon platform. The same problem occur when memory intensive codes are scaled to two processors. As the cache size is close to test system 1 / Intel Xeon the blocking factor stayed constant and provides best performance at a size of 28 as seen in figure 3.3. For the Itanium2 the Intel Fortran compiler 9.0_024 Build 20050624 was used [4]. The compiler flags *-O3*, *-ivdep_parallel* and *fno-alias* were used for best performance.

3.2 Comparison of two different implementations

In order to compare the two algorithms one should look at the differences first. Although *LBMKernel* and *LBM2BC* are both full matrix (grid) based algorithms, the skewed channel of the *LBM2BC* implementation reduces the amount of fluid cells while it increases the need of the more complex Bouzidi's et.al. [6] bounce-back interpolation boundary treatment. For a first comparison some basic benchmarks without the incline and the sphere features of the *LBM2BC*, thus a complete empty channel, were evaluated. The corresponding *IJKL* layout (propagation optimized) of the *LBMKernel* was chosen and the unstructured list with LIJK layout. For the *LBM2BC*, the standard implementation with full matrix and IJKL layout and the variant with unstructured list implementation and LIJK layout were chosen.

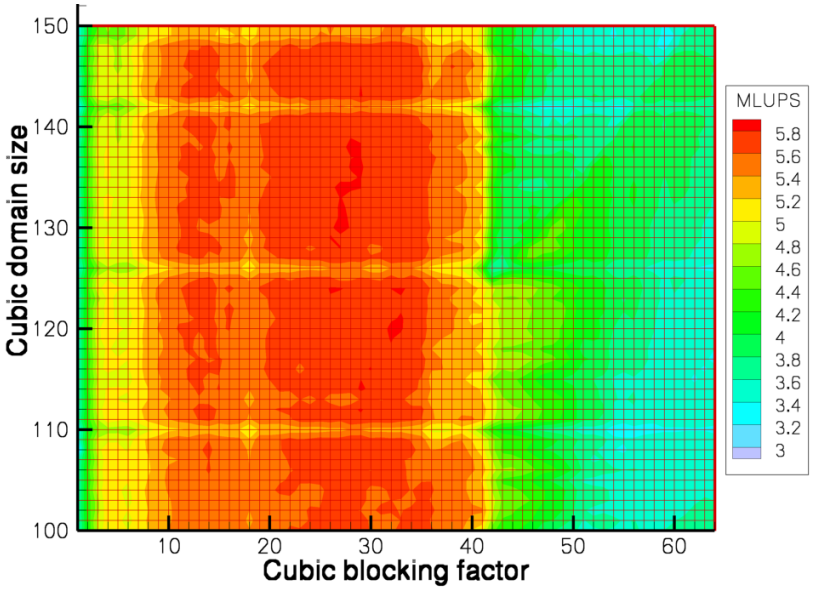


Figure 3.3: Parameter study of blocking factors for the unstructered *LBM2BC* list layout LIJK on test system 3 / Intel Itanium2.

System	Platform	Clock speed	Sockets	Cores / socket
1	Intel Xeon IA32 EM64T	3.6 GHz	2	1
2	AMD Opteron IA32 AMD64	2.2 GHz	4	1
3	Intel Itanium2 IA64	1.4 GHz	2	1

Table 3.1: Test system overview.

System	Caches	Memory	Memory bandwidth	Memory access	Execution model
1	2 MB L2	4 GB	5.3 GB/s PC2700	UMA	Out-of-order
2	1 MB L2	8 GB	6.4 GB/s PC3200	ccNUMA	Out-of-order
3	1.5 MB L3	10 GB	6.4 GB/s PC3200	UMA	EPIC

Table 3.2: Test system details.

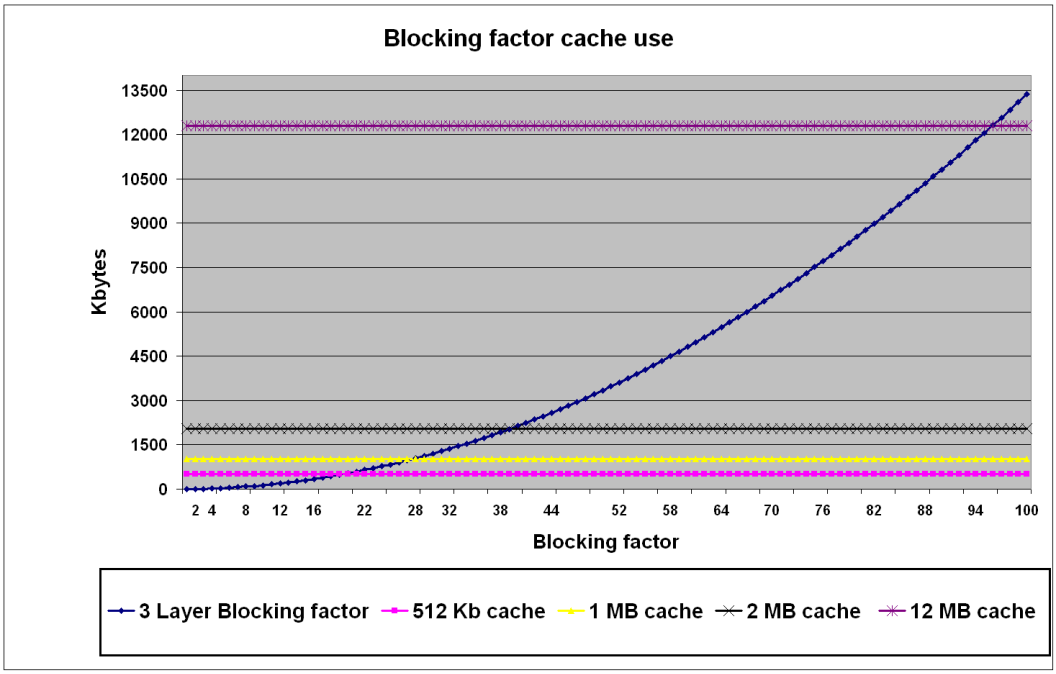


Figure 3.4: Memory / Cache usage of different blocking factors. Additionally the cache sizes of different test systems is shown.

3.2.1 Test system 1 / Intel Xeon

Caused by the additional computations the *LBM2BC* boundary treatment demands much more computational effort. As a consequence the *LBM2BC* was expected to perform worse than the *LBMKernel*. The benchmarking result on the first platform, Intel Xeon, seen in figure 3.5 however show non confirming results. First a calculation by the rule of thumb provides the knowledge how much data is actually needed. The most consuming part of the algorithm is the particle distribution array *pdf*. This *pdf* has cubic dimensions of 16 for example. For each grid point 19 velocities are accessed in the collision for two time steps, namely for the current and next timestep. Additionally 8 bytes are accounted for each double precision number. This sums up with:

$$16 * 16 * 16 * 19 * 2 * 8 \text{ Bytes } = 1,216 \text{ MB}.$$

At test system one, using 2 MB of L2 cache, this calculation will run with cache performance. All succeeding grid sizes will exceed the cache and so perform much slower. Coming to reasonable grid sizes of 64 x 64 x 64 and larger, the performance should drop down to the actual memory performance. As seen in figure 3.5 the *LBM2BC* code outperforms the grid implementation of the *LBMKernel*. Comparing the two unstructured list based implementations the same relation is seen. The performance gains are only moderate but measurable. Due to the expectation, that the *LBM2BC* has to be outperformed by the *LBMKernel* a closer look into the execution times of the two main computation routines was done. The measured fractions of total execution time can be seen in figure 3.6 for the *LBM2BC* implementation and in figure 3.7 for the *LBMKernel*. The interesting fact is the relation between *bounce-back* and

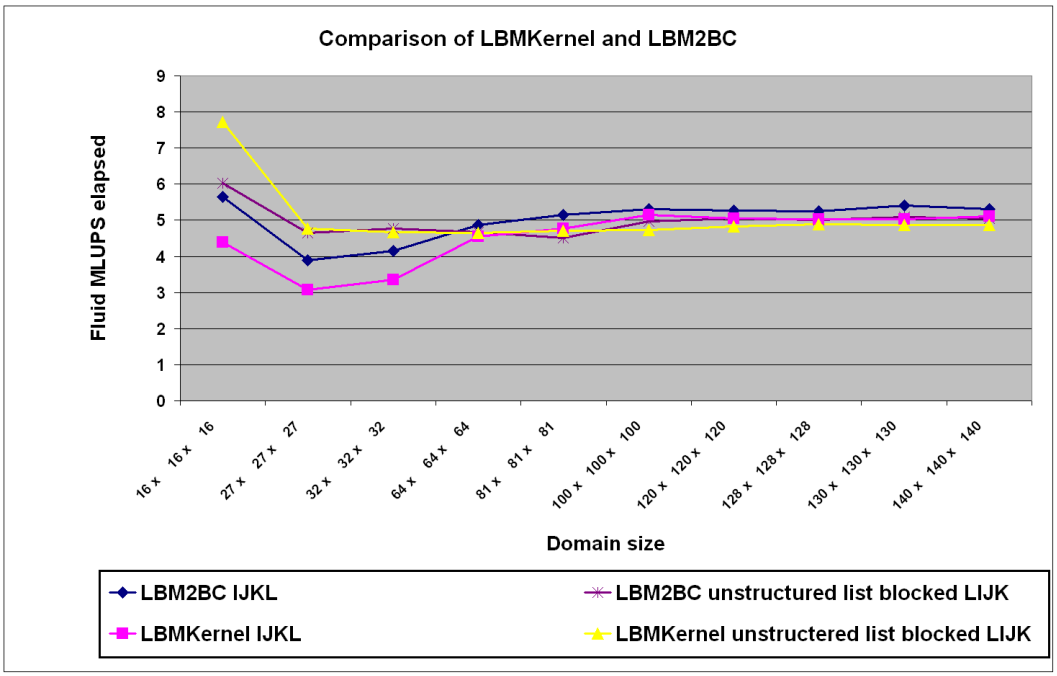


Figure 3.5: Comparison of fluid MLUPS on test system 1 / Intel Xeon.

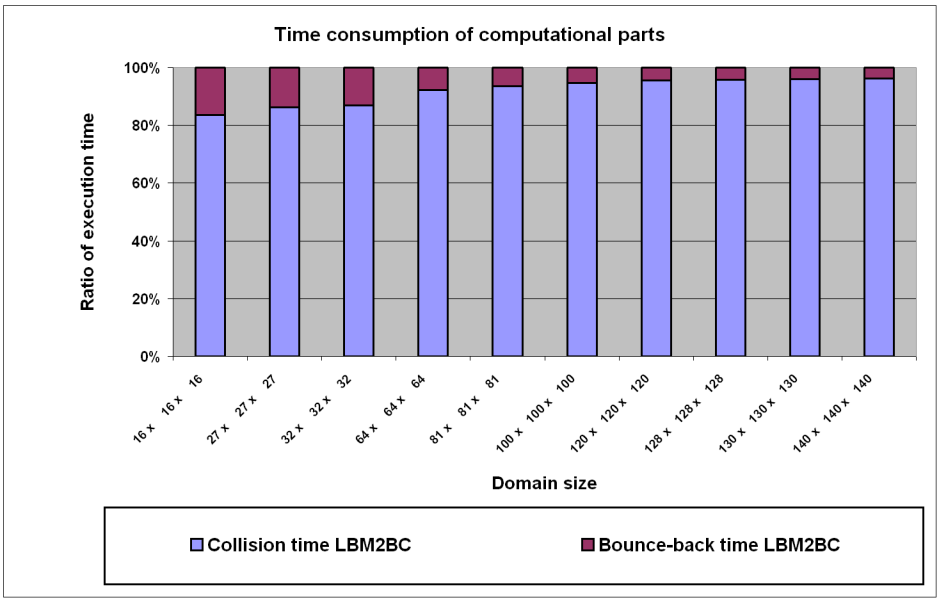


Figure 3.6: Comparison of execution time of *LBM2BC* on test system 1 / Intel Xeon.

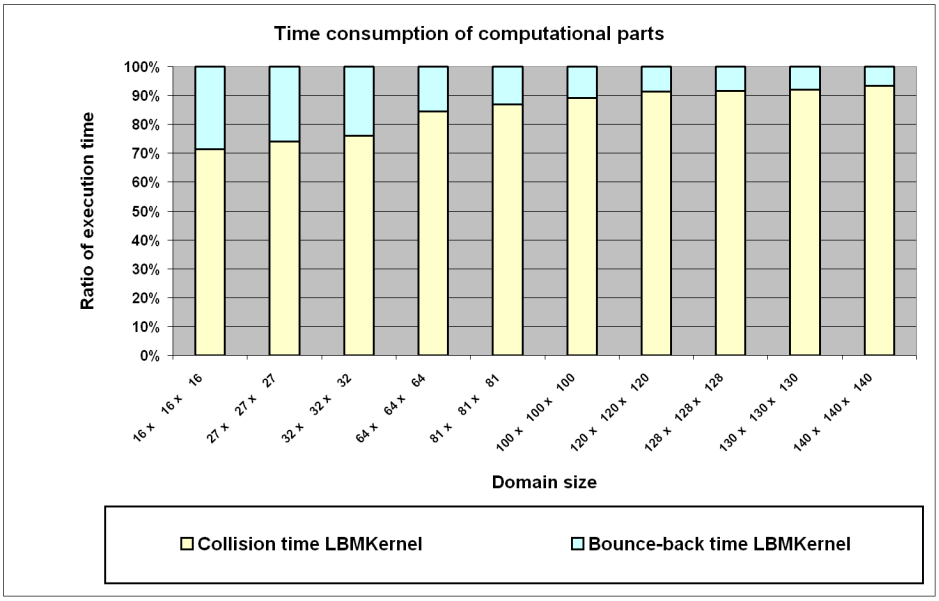


Figure 3.7: Comparison of execution time of *LBMKernel* on test system 1 / Intel Xeon.

collision calculation. The *LBMKernel* almost spends 10 to 25 percent of its runtime inside the *bounce-back* routine. In contrast, the *LBM2BC* does not exceed 5 percent of computation runtime in most cases. A close comparison between the bounce-back routine of *LBM2BC* and *LBMKernel* revealed the behavior. The additional computations were the most obvious change within the calculation. Thinking of the above mentioned memory bandwidth problem regarding the LBM algorithms, these additional computations may have, platform dependent, no impact on performance at all. The former benchmarks of the LBM algorithms always showed a performance substantially less than the possible processor peak performance ([19]). As a conclusion the processor waits for actual calculation while the memory hierarchy is trying to get the needed data. The only possible performance gain can be achieved by reducing the data needed for the calculation. Remembering the results of section 2.1, the *LBMKernel* copies for one obstacle cell, 19 velocities from and to the cell itself and the adjacent cells. In comparison to that, the preprocessing routines of the *LBM2BC* determines which links of a fluid cell touch or intersect the boundary wall. Only these links are then treated with the Bouzidi bounce-back interpolation algorithm. Thinking of a normal flat channel this would be a maximum of 5 links. All other links are treated as normal fluid cells in the *collision* routine or are obstacle links which need no attention. This is of course only the benefit of this altered calculation. Additionally a huge management algorithm and more management data arrays must be considered. The result is an algorithm which maintains the second order boundary discretization but also enhances the performance.

The list layout [10] is natively not as sensitive to cache effects, as the grid layout, because it operates on an unstructered list where the indirect addressing hides data dependencies. Though the indirect addressing should degrade performance and the management of the additional list uses cache and memory bandwidth the performance degradations are only minor. Hence, the special blocking algorithm applied reduces these negatives effects as it is increasing spatial data locality without any alteration of the list collision algorithm.

3.2.2 Test system 2 / AMD Opteron

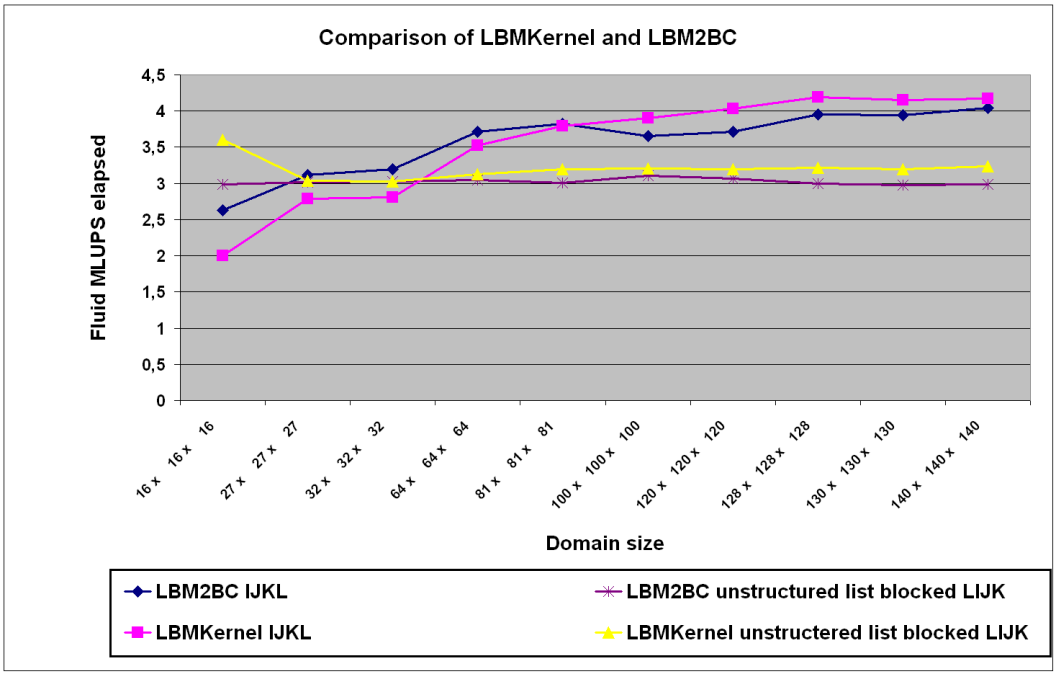


Figure 3.8: Comparison of fluid MLUPS on test system 2 / AMD Opteron.

For the AMD Opteron based test system, the performance of the different kernel can be seen in figure 3.8. Most obvious the performance substantially fluctuated in comparison to the Intel Xeon platform, where the best and worse performance were only 0.4 MLUPS apart. Here the gap is about 1.6 MLUPS. The best performance dropped from 5.3 MLUPS at grid size 140 x 140 x 140 to 4.2 MLUPS. The *LBMKernel* performs only slighter better than the *LBM2BC*. The list layout based algorithms are substantially slower and perform both nearly constantly above the 3 MLUPS marker. Like the grid based variants they switched and the *LBMKernel* based list layout performs better. The L2 cache of test system 2 / AMD Opteron is only 1 and has a lower transfer bandwidth. As a consequence overall performance drops down.

3.2.3 Test system 3 / Intel Itanium2

The third test system is an Intel Itanium2 dual node. As mentioned in section 3.1 the Intel Itanium2 represents a completely different platform. The performance for any application depends mostly on the compiler quality and its ability to optimize the code. Again, the *LBMKernel* and *LBM2BC* perform close together, the *LBM2BC* always a little better. However the difference is less than on the IA32 based platform, as the compiler prefetches decreased the performance cost of the bounce-back memory operations described in section 3.2.1. The grid based layouts operate both at about 8.5 MLUPS which outperforms the other test systems. The list based *LBMKernel* code performs at about 6.5 MLUPS. Again the Bouzidi boundary conditions with interpolation claim their tribute and give the *LBM2BC* based list

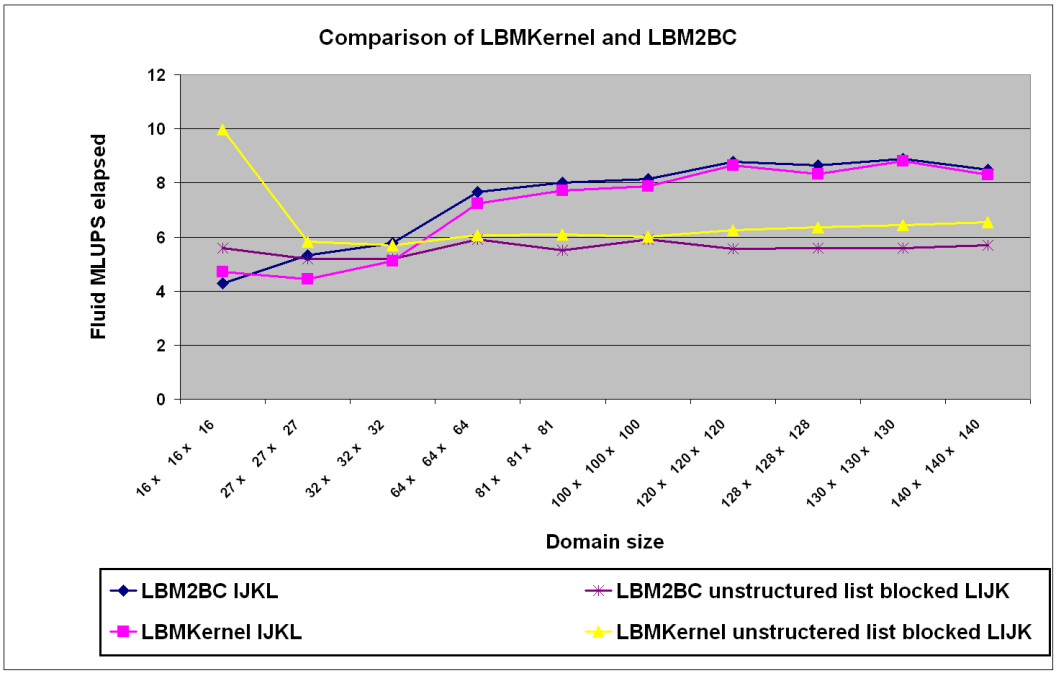


Figure 3.9: Comparison of fluid MLUPS on test system 3 / Intel Itanium2.

variant the worst performance with about 5.6 MLUPS. In comparison to both IA32 architectures, the gap between grid and unstructered list implementations has increased. With the *-opt_report* flag the Intel Fortran EM64T 9.0 compiler [4], offers a detailed view inside the conducted compiler optimizations. Of especial interest on the IA64 test system is the amount of prefetches. The grid based *LBM2BC* has a overall prefetch of 39 streams. This complies with the estimations of 19 velocities in two time steps. The unstructered list based algorithm shows a prefetch of only 6 streams, which of course is substantially less than the grid version and causes the 2 MLUPS less performance

3.3 Additional blocking factor parameter studies

Nowadays a large range of on-chip cache sizes ($1-12$ MB) is available and the blocking factor has to be chosen with care. To demonstrate the sensitivity of performance and optimal blocking factor to cache size, the blocking factor parameter study was extended to three further test systems 4,5 and 6. Detailed specifications can be seen in table 3.4. Comparing the block memory usage estimated in equation 2.6 to the performance results from test system 1 / Intel Xeon shown in figure 3.1, it is evident that blocking factors exceeding the cache have less performance. The transition to the largest blocking factors, theoretically still fits into the cache. However, as the cache is nearly full other required data, cache associativity conflicts and other performance degrading effects decrease overall blocking benefit.

Test system 4 / Intel Xeon offers the smallest cache size and the optimal blocking factor dropped down between 8 and 18 as seen in figure 3.10. The performance break in at grid

System	Platform	Clock speed	Sockets	Cores / socket
4	Intel Xeon IA32	2.6 GHz	2	1
5	Intel Xeon IA32 EM64T	3.2 GHz	2	1
6	Intel Itanium2 Montecito IA64	1.4 GHz	2	2
7	AMD Opteron AMD64	2.2 GHz	4	2

Table 3.3: Parameter study test system overview.

System	Caches	Memory	Memory bandwidth	Memory access	Execution model
4	512 KB L2	4 GB	4,2 GB/s PC2100	UMA	Out-of-order
5	1 MB L2	8 GB	5,3 GB/s PC2700	UMA	Out-of-order
6	12 MB L3	10 GB	6.4 GB/s PC3200	UMA	EPIC
7	1 MB L3	16 GB	6.4 GB/s PC3200	UMA	Out-of-order

Table 3.4: Parameter study test system details.

sizes of 126 x 126 x 126 is related to the power of two cache thrashing effect. The true domain size including the ghost layers is 128 x 128 x 128, thus a power of two. Similar effects can be seen in all other parameter study test, but not as distinctive, as the larger cache sizes have a higher associativity. Limited by the systems RAM memory, benchmarks to a grid size of 150 x 150 x 150 were not possible.

Figure 3.11 shows that test system 5 / Intel Xeon with 1 MB cache has the best performance between blocking factors of 18 and 28. The main difference between test system 1, 4 and 5 are the amount of cache.

The test systems 3 and 6 are both Itanium2 platforms. Test system 6 is equipped with the most current Intel Itanium2 dual core design, codename "Montecito" having very large on-chip caches of 12 MB for each core. The test system 3 / Intel Itanium2 shows the best performance in a broader range of blocking sizes from 8 to 42 in comparison to the Intel Xeon based test systems. With the large L3 cache of test system 6 the blocking factors increased and range from 50 to 80 in grid sizes between 100 x 100 x 100 and 120 x 120 x 120. For larger systems up to 150 x 150 x 150 a blocking factor between 20 and 90 provides best performance.

3.4 Performance impact for complex geometries

The previous section 3.2 presented the performance of the *LBM2BC* on flat and open channels. The development however, aims to simulate more complex surfaces. The following charts will present the performance behavior of the *LBM2BC* according to changing surfaces. Over the whole execution the domain sizes change from 32 x 32 x 32 to 196 x 196 x 196. For every grid size the FlowOpening is reduced from an fully open channel to a minimum of half the channel height. This has an impact on the overall fluid cell count and onto the

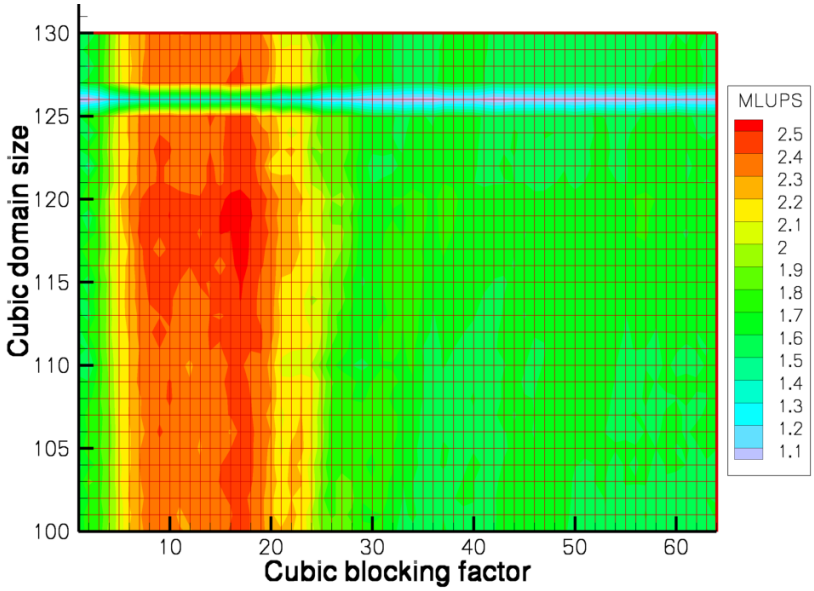


Figure 3.10: Parameter study of blocking factors for the unstructered *LBM2BC* list layout LIJK on test system 4 / Intel Xeon.

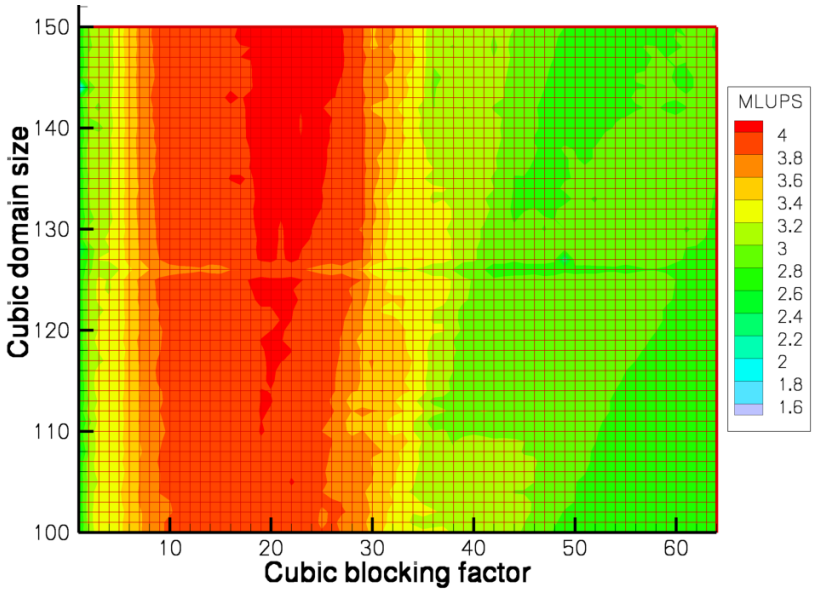


Figure 3.11: Parameter study of blocking factors for the unstructered *LBM2BC* list layout LIJK on test system 5 / Intel Xeon.

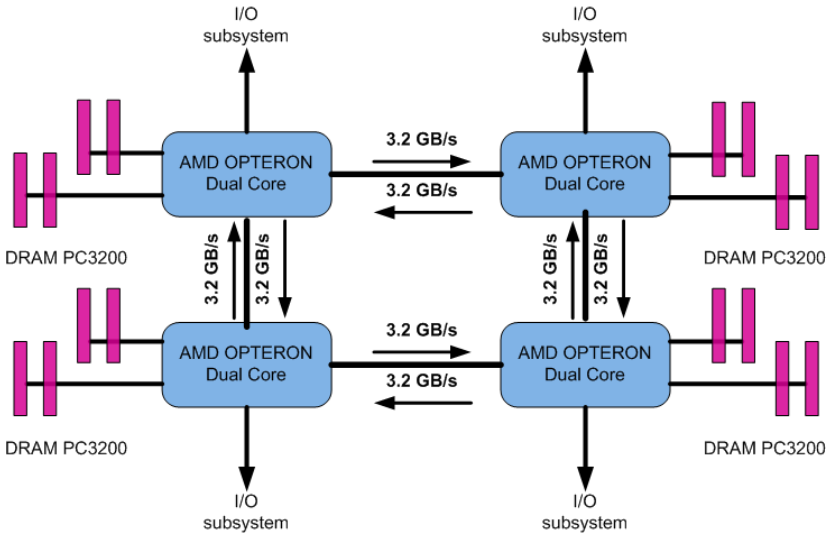


Figure 3.12: Schematic AMD Opteron dual core quad node interconnect.

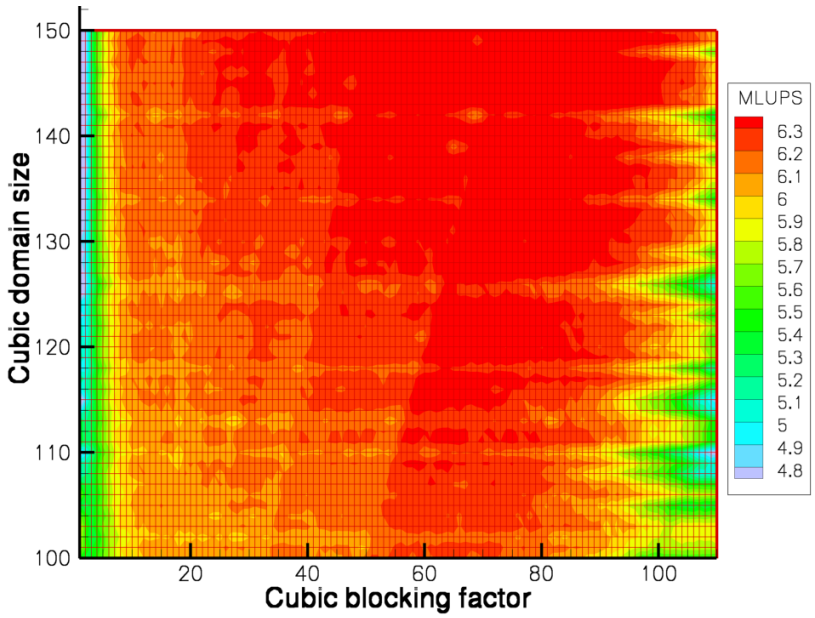


Figure 3.13: Parameter study of blocking factors for the unstructered *LBM2BC* list layout LIJK on test system 6 / Intel Itanium2.

count of cells which need a boundary treatment. The maximum of the cut links was 5 with the flat channel. This count is now increased to a maximum of eight links which must be cut. As a consequence the Bouzidi's et.al. [6] bounce-back boundary conditions with interpolation demand more computations and memory transfers. Please note, that the memory requirement of the list is decreased linearly with a decreasing *FlowOpening*, while it stays constant for the full matrix based implementation.

3.4.1 Test system 1 / Intel Xeon

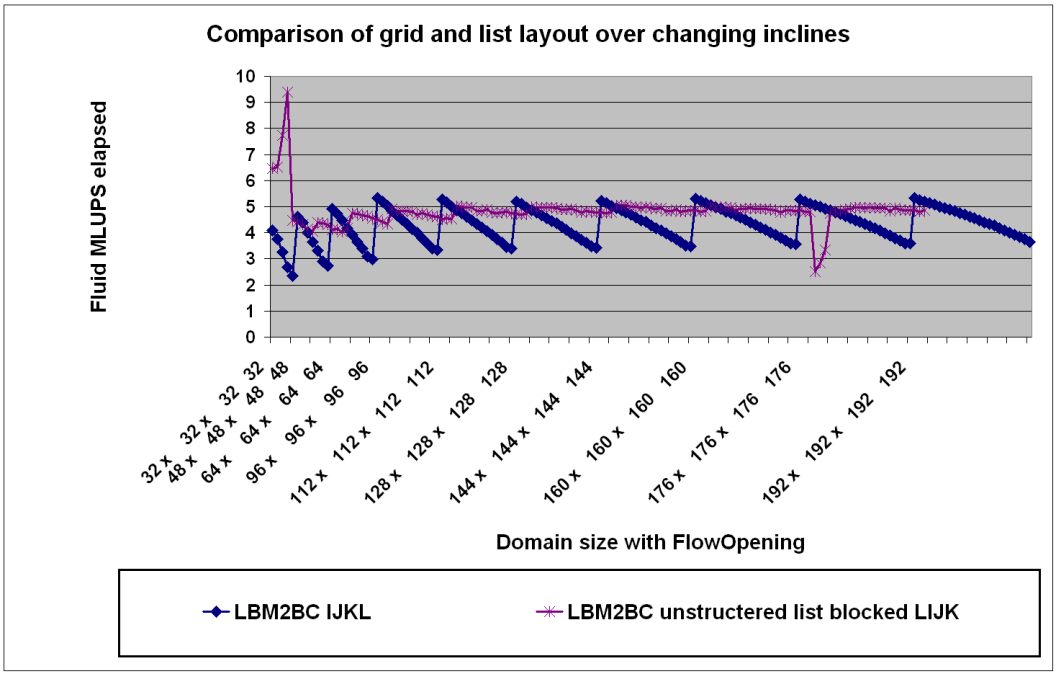


Figure 3.14: Comparison of fluid MLUPS over different inclines on test system 1 / Intel Xeon. For every domain size, the flow opening starts with an full opened channel (*FlowOpening* = *kEnd*) and is decreased to the half of *kEnd*. The x-axis shows the three domain directions i,j,k and the initial *FlowOpening*.

To present the performance an uncommon chart format was chosen, which increases the domain size in the first place. Between each domain enlargement, the *FlowOpening* is reduced from a fully open channel to an half open inclined channel. The start of an increased grid size is marked through the x-axis label. At that point we have the same performance results as in section 3.2, as the channel is still fully opened. Apart from the peaks at the initial grid size 32 x 32 x 32, which are affected by cache performance of the small grid size, the *LBM2BC* list implementation performs little below 5 MLUPS over all domain sizes and inclines. This confirms the assumption that the unstructered list layout does not depend onto the channel layout, as it only stores the fluid cells but not the dispensable obstacle cells [10]. The grid layout starts with an fully opened channel at its peak performance with 5.4 MLUPS and drops then down to 3.5 MLUPS with half *FlowOpening* and larger grid sizes as seen in figure 3.14. The first expectation would be, that with decreasing fluid cells to calculate on, the amount of memory which has to be transfered and worked on now, decreases as well. At a

closer look the access is now, non contiguous. The processor fetches data not each element on its own, but in cache lines. These cache lines are usually 16 double words in size. This means, the memory subsystem has to transfer all items of one cache line into the cache, regardless how much elements are accessed. With an open channel, the algorithm runs over the succeeding data elements, which are already fetched. In the case of the inclined channel however, it happens that these cache lines end within elements, which are never calculated (fixed obstacles). With an growing incline these fixed obstacle hits will increase too, as the overall boundary area increases. As a consequence the performs drops with rising incline. The current list implementation for benchmark purposes is based on an prepopulated grid. This initialization process needs the double memory size at the initialization phase. Unfortunately test system 1 / Intel Xeon has only 4 GB of main memory and cannot host the large 192 x 192 x 192 domain calculations in list mode and there are no results from the *LBM2BC* list implementation. The list layout is outperformed by the grid layout only with the fully open channel and some slightly inclined channels. For a *FlowOpening* which is less than 80 percent of the channel height, the list layout performs better than the grid based layout. Most notably we are able to achieve almost the same performance, even though additional data is required for the list management and the indirect addressing covers data dependencies.

3.4.2 Test system 2 / AMD Opteron

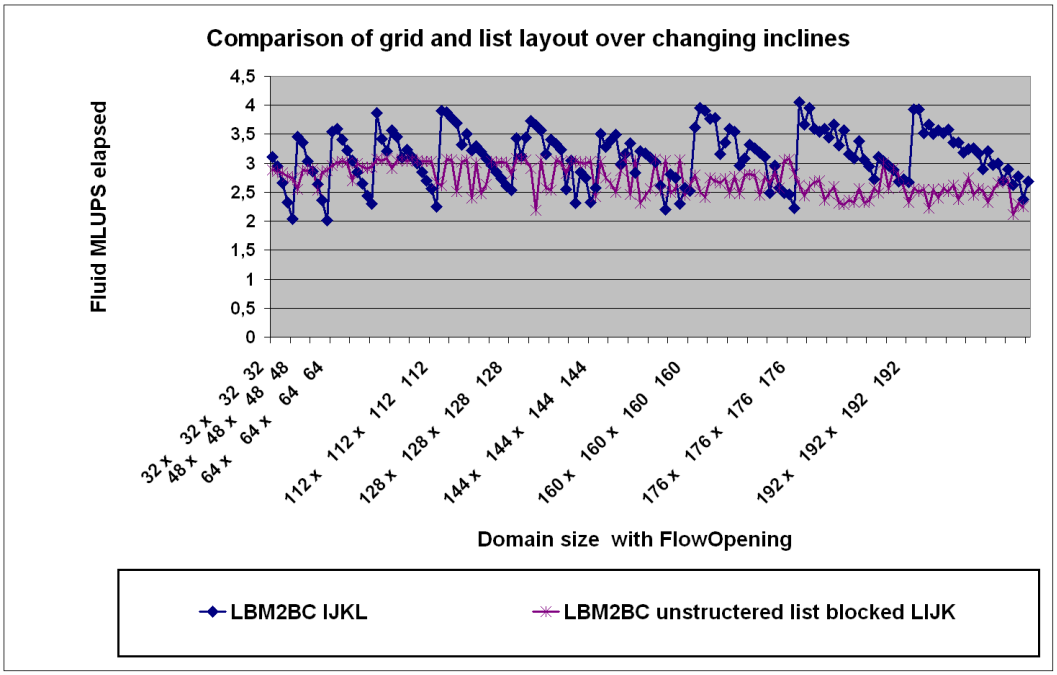


Figure 3.15: Comparison of fluid MLUPS over different inclines on test system 2 / AMD Opteron. For every domain size, the flow opening starts with an full opened channel (*FlowOpening* = *kEnd*) and is decreased to the half of *kEnd*. The x-axis shows the three domain directions i,j,k and the initial *FlowOpening*.

On the first glance, the benchmark results on test system two, seen in figure 3.15 are more fluctuating than those on test system one. This is a result of the to 1 MB reduced L2 cache,

where less data could be hold in cache. Additionally the overall four paths to different memory locations can influence the results. Despite these interfering factors the overall character is the same as seen on test system 1. The *LBM2BC* grid layout starts at the peak performance of 4 MLUPS, known from section 3.2 and drops down to 2.2 MLUPS with half open channel. The list layout performs about 3 MLUPS till a domain size of 144 x 144 x 144 and then drops down to 2.2 at 192 x 192 x 192. Despite the non smooth measurements of the list layout, it mostly outperforms the grid layout if the *FlowOpening* drops below 60 percent of the channel opening.

3.4.3 Test system 3 / Intel Itanium2

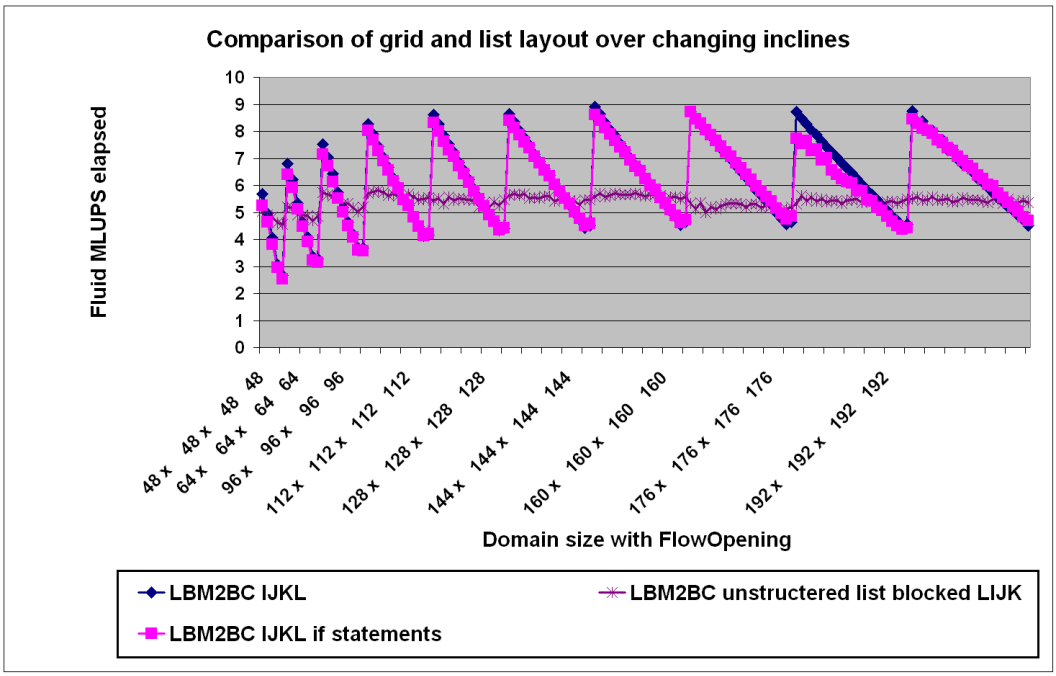


Figure 3.16: Comparison of fluid MLUPS over different inclines on test system 3 / Intel Itanium2. For every domain size, the flow opening starts with an full opened channel (*FlowOpening* = *kEnd*) and is decreased to the half of *kEnd*. The x-axis shows the three domain directions i,j,k and the initial *FlowOpening*.

Test system 3 / Intel Itanium2 shows superior performance with the *LBM2BC* again at about 8.9 MLUPS with a fully open channel in figure 3.16. With increasing FlowOpening however, the performance drops down to 50 percent unaffected by the grid size. As described in the preceding subsections, the inclined channel compromises the continuous prefetching but never forced the performance to drop below 50 percent of the open channel performance. Section 3.1 described the dependence of the Intel Itanium2 to the quality of optimized code. In this case, the code is optimized for prefetching but with increasing incline a decreasing fraction of the prefetched data is used. This causes the high loss of performance while maintaining peak performance with an open channel on all layouts.

The algorithm used as a basis was optimized to run on open channels, where the count of

obstacles is negligible in comparison to the fluid cell count. However with the inclined channel layouts, this assumption no longer holds. To take advantage of the Itanium2 compiler optimizations the *if* statements inside the *collide-stream* step were taken out ([9]). These *if* statements determine three different operational blocks. The first one calculates the macroscopic quantity of the cell itself, this is a read only operation. The other two blocks calculate and write to the adjacent cell, thus read and write. As a consequence the first block can be uncommented without any numerical difference. However the compiler now has a loop without a branch, which is an advantage inside the program execution and the additional arithmetic operations and data transfers did not circumvent a performance gain. With rising obstacle count this additional memory read operations, however increase too. The *if* statements removed algorithm produces little better results which can be seen in figure 3.16 when the channel is completely open. With the half closed channel both implementations perform equally. As a consequence the performance gain yielding from the vectorization of the *if* statement free code is as high as the performance cost of the additional operations and the higher memory load on inclined channels. The list layout stays at a constant performance of about 5.3 MLUPS and has only fluctuations at small domain sizes. A benefit from the unstructered list layout is only gained, if the FlowOpening is less than 60 percent of the channel height.

Chapter 4

Performance of simple parallelizing approach

Shared memory computing nodes become more and more powerful and are widely used in scientific and technical computations. Additionally the development goes to dual and multi core CPUs, which increases the medium grained parallelism. To regard this, section 4.1 presents the implementation, performance and scalability of the full matrix based *LBM2BC* algorithm utilizing OpenMP

4.1 OpenMP implementation

On shared memory systems, a simple way of parallelizing a program is OpenMP [1]. OpenMP in general extends high level programming languages with so called OpenMP pragmas (e.g. *!$OMP*). The main idea of OpenMP is to parallelize do loops (e.g. *!$OMP PARALLEL DO*).

All variables in the parallel section are shared per default. This means, that there is one common variable for all working threads. The default case of shared variables can easily be altered. This is very comfortable to get OpenMP quickly started, but in case of complex programs this could be a performance issue and even affect correctness of calculations. In the single threaded program version of *LBM2BC* we introduced temporary variables to calculate the macroscopic quantities in the *collide–stream* step. If these variables are shared and the threads perform their read and write operations concurrently, the result is corrupted, due to race-conditions. Thus, these variables have to be declared as private such that each thread holds its own private copy of them. Note that private variables have an undefined state after exiting the parallel section. Results, which should be accessed after the parallel loop, require to be stored in shared variables.

As a basic rule of OpenMP one should always parallelize the outermost loop to give each thread as much time as possible to calculate without synchronization, as each ending *!$OMP END PARALLEL DO* has an implied barrier. This was also done with the *LBM2BC* with respect to special private and shared variables. The domain is decomposed along the height dimension, meaning that the k-axis is split amongst the threads. Regarding the two grid layout of the *LBM2BC* , there are no concurrent read and write operations to the *pdf*, as all threads read from one timestep while writing is performed to the other, without any overlap among the write operations.

On shared memory nodes with physically distributed memory, e.g. AMD Opteron, another problem has to be considered. The distributed memory allows shared access to all memory locations, under the restrain of higher latency and lower bandwidth, from or to remote

locations. The data placement strategy decides in which physical memory the data finally ends up. A general strategy is called first touch, meaning that memory is allocated at the nearest possible location to the processor, which performs the first write access. Hence, to pass the correct domain decomposition to the memory hierarchy, the initialization of the algorithm has to be parallelized exactly the same way as the calculation routine.

The standard OpenMP parallelized *LBM2BC* now has the standard *!$OMP PARALLEL DO* pragma before the collision routine's *k*-loop. Additionally the allocation of the memory was also parallelized as described above. In contrast to the expectations, this simple approach results in a worse performance on Intel Itanium2 based systems. A detailed analysis of the compiler optimization output showed (compiling with *-opt_report*), that switching on the OpenMP pragmas, restrained the compilers optimization options. The most impact on the performance had the decreased prefetch count inside the *collide–stream* subroutine. As a consequence the OpenMP pragma was removed, also implicating no further parallelism.
In order to parallelize this phase of the calculation again, the subroutine call inside the *main* subroutine was enclosed by *!$OMP PARALLEL* pragmas and the calculation inside the *collide–stream* subroutine was manually parallelized as seen in algorithm 4.1.

This modified implementation, showed with one thread the same performance as the version without OpenMP. Thus the compiler applied the same optimizations and the OpenMP pragmas no longer had a negative influence on the execution.

Algorithm 4.1: OpenMP manual scheduling of collide-stream step

```
1  #ifdef OMP
2  !Getting information for local thread
3
4  myid = OMP_GET_THREAD_NUM()
5  nthreads = OMP_GET_NUM_THREADS()
6
7  stride=UBOUNDK/nthreads
8  OMPLBOUNDK= 1+   myid*stride
9  OMPUBOUNDK=(myid+1)*stride
10
11 if(myid == 0)then
12       OMPLBOUNDK=1
13 endif
14
15 !Last thread gets cleanup work
16
17 if(myid == (nthreads-1))then
18       OMPUBOUNDK=kEnd
19 endif
20
21 do k=OMPLBOUNDK,OMPUBOUNDK, LoopDir
22
23 #else
24 !If compiled without OpenMP
25
26 do k=1,KEnd, LoopDir
27
28 #endif
```

4.2 OpenMP performance

In section 4.1 the implementation of the OpenMP pragmas and routines into the *LBM2BC* was described. The parallelization of an algorithm contains a lot of difficulties and requires a deep understanding of the data dependencies. It is not unusual, that an algorithm, although running in parallel with more computing power, gets even slower. As mentioned in section 4.1 there are even some unfamiliar problems which occur. The consequence is, that the parallelized code has to be checked not only for miscalculations but also on performance degradations. In general the *LBM2BC* is a very memory intensive code [14]. This may lead to minor performance improvements than expected, if the CPU's share one common path to the memory. In the following the performance measurements on the test systems are presented. For better comparison the performance measurements of the standard sequential *LBM2BC* were added. In order to compare the speedup which is gained by running with two CPUs, the same executable was tested with only one CPU, meaning with *OMP_NUM_THREADS = 1*. Note that since HyperThreading was disabled, the number of threads is equal to the number of CPUs being used.

4.2.1 Test system 1 / Intel Xeon

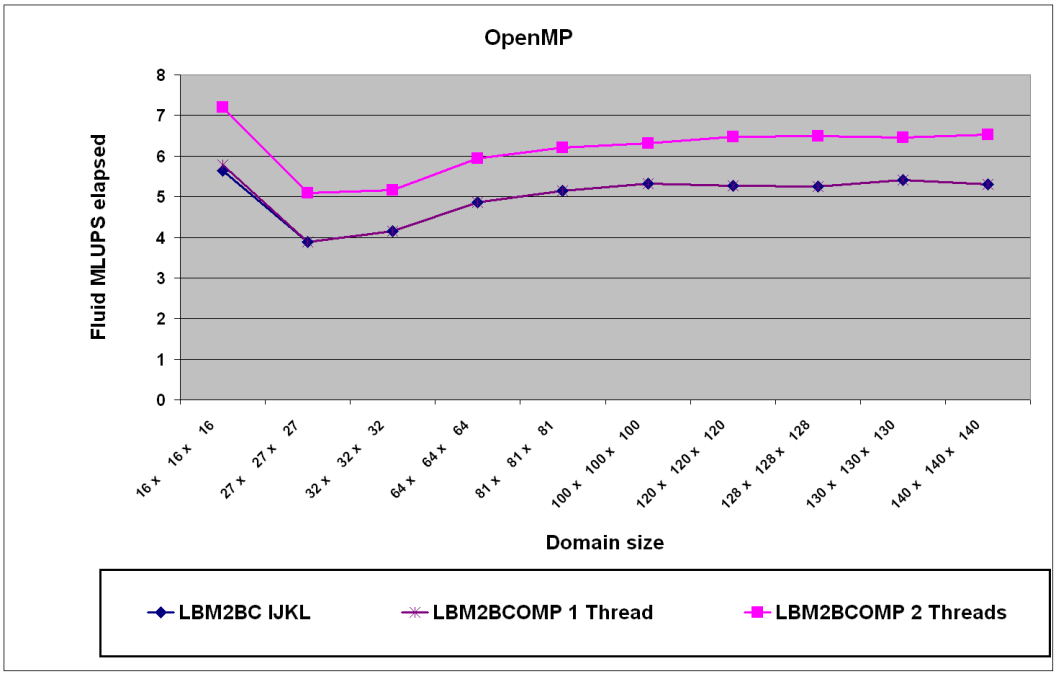


Figure 4.1: Comparison of fluid MLUPS on test system 1 / Intel Xeon.

The performance as seen in figure 4.1 is the same for the sequential and OpenMP with 1 thread. This is the benefit of the manual OpenMP thread scheduling, described in section 4.1. Using two threads a speedup of about 20 percent could be achieved.

Obviously the memory bus represents the bottleneck on this platform, with this current data

layout and access pattern. A theoretical view at the *LBM2BC* algorithm reveals that the collision routine is responsible for over 90 percent of computational time. The Intel Xeon processor has a clock speed of 3.6 GHz and a memory subsystem with a maximum transfer rate of 5.4 GB per second. For each cell we need to access 19 of its own variables and write to 19 corresponding links in the next timestep grid. Additionally the write misses increase the memory transfers by 50 percent. This sums up to 456 bytes of data per lattice cell. Concerning the maximum transfer rate of 5.4 GB/s a maximum memory performance of $\approx$ 12 MLUPS as stated in [19], can be achieved. Considering that one cell update needs a maximum of 200 cycles inside the CPU core we have a maximum of

$$\frac{3.6\,\mathrm{Gcycle}}{200\,\mathrm{cycle}\,/\,\mathrm{MLUPS}} = 18\,\mathrm{MLUPS}. \tag{4.1}$$

Together these two estimations, show that one processor most likely can process all data available from the memory interface. However the two thread OpenMP code still has a performance of just 20 percent better than the one thread version. Thinking of the second CPU we not only use the additional arithmetic units but the cache too. The increased cache size for the same problem and a slightly better usage of the memory bus causes the speedup.

4.2.2 Test system 2 / AMD Opteron

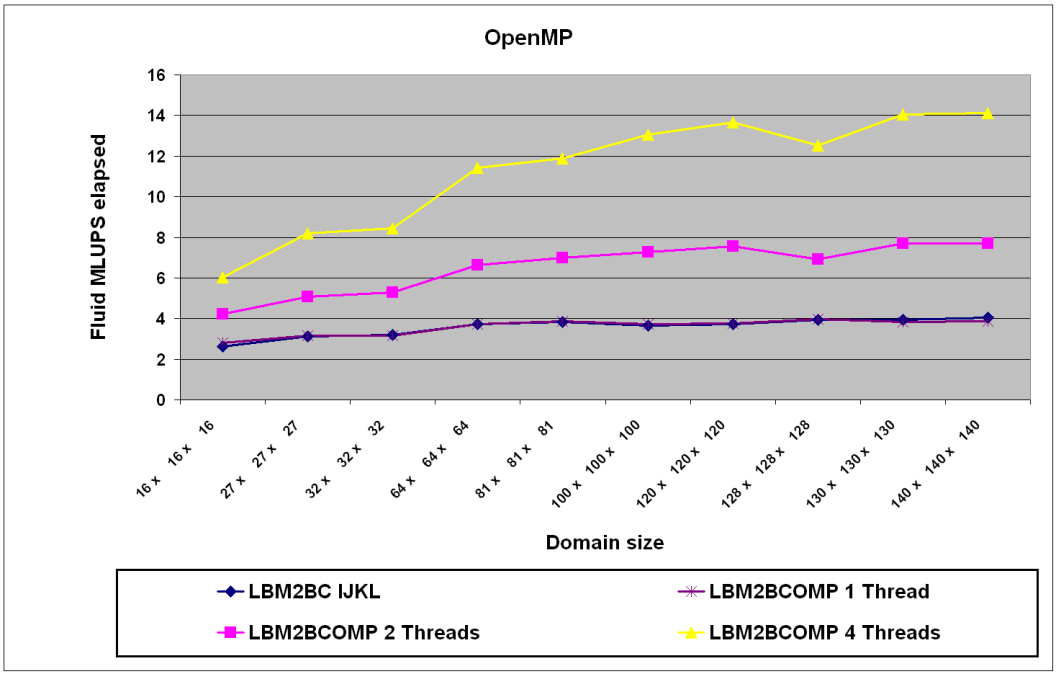


Figure 4.2: Comparison of fluid MLUPS on test system 2 / AMD Opteron.

In contrast to test system 1 / Intel Xeon, the test system 2 / AMD Opteron based quad node has 4 independent paths to the memory. Theoretically this should allow for a perfect parallel speedup. In figure 4.2 is seen that the serial version performs equally to the parallelized variant with one processor core. This provides a common frame of reference.

Indeed the performance gain has great improved compared to test system 1 / Intel Xeon, switching from one to two processors cores. In contrast to test system 1 / Intel Xeon the algorithm has the double memory bandwidth available for use. Distributing the work amongst 4 processors scales perfectly to about 14.1 MLUPS. The conclusion from this results and the moderate speedup on test system 1 / Intel Xeon, shows that the *LBM2BC* is a very memory intensive code and does not depend much on the pure arithmetic power of the processor, at least for large domains. As it is very interesting how this platform would behave if we add more processors but not increasing the memory bandwidth we also benchmarked on the additional test system 7 / AMD Opteron dual core as seen in tables 3.3, 3.4 and figure 3.12. This simulates the behavior of test system 1 / Intel Xeon, as the arithmetic units and the cache is doubled, while the memory bandwidth stays constant. Note that first each dual core processor gets one thread to work.

As seen in figure 4.3 the scaling from one over two to four processor cores is as good as on test system 2 / AMD Opteron. The overall higher performance is caused by the DDR 3200 memory interface in contrast to DDR 2700 on test system 2 / AMD Opteron. Till four processor cores, each executing processor got its own memory bus. The increase to eight processor cores will force each first used processor core to share its memory bus with the second core on its socket. The performance at about 20 MLUPS with eight processor cores compared to 16 MLUPS with 4 processor cores shows that the memory bus was not fully utilized by four processor cores and that the increased overall cache size is an advantage. However the speedup decreased from about 82 percent to 25 percent.

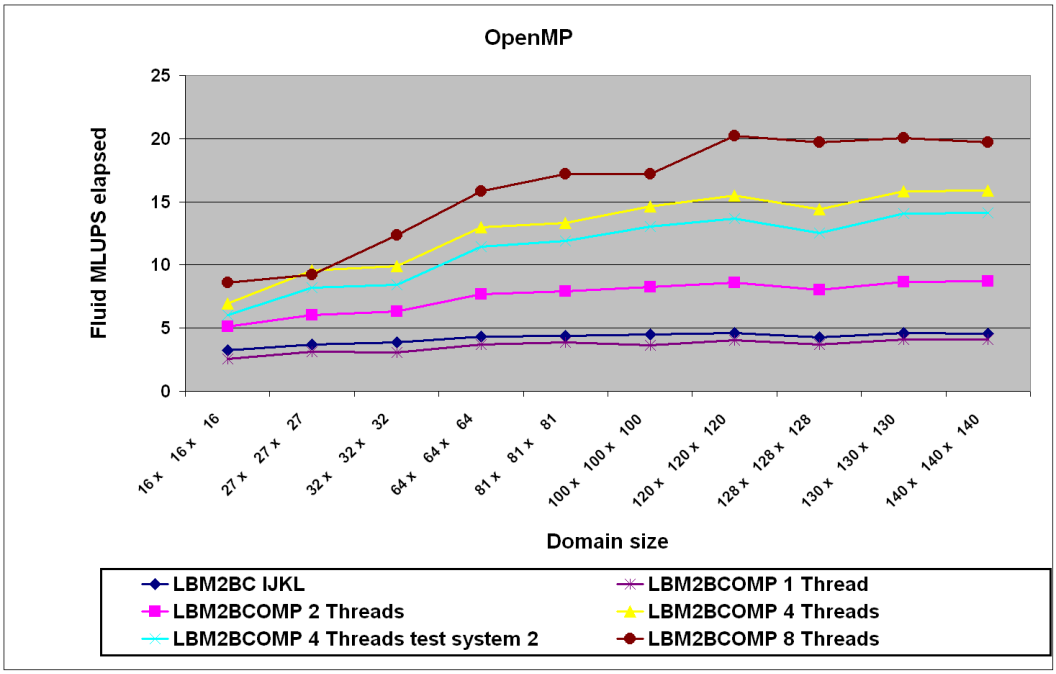


Figure 4.3: Comparison of fluid MLUPS on test system 7 / AMD Opteron.

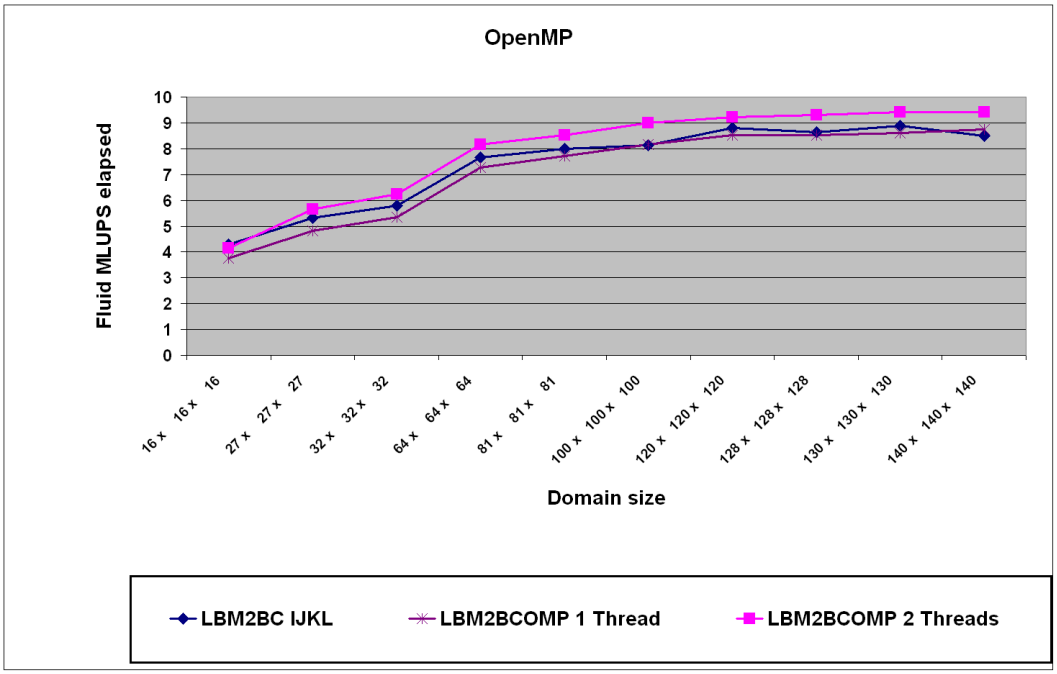


Figure 4.4: Comparison of fluid MLUPS on test system 3 / Intel Itanium2.

4.2.3 Test system 3 / Intel Itanium2

Test system 3 shows the anomaly that the pure existence of OpenMP slows the *LBM2BC* down as described before. The enhanced parallel algorithm performs equal to the OpenMP missing variant. Regarding the one thread performance we get improvements of only 8 percent. This leads to the conclusion, that one CPU already completely utilizes the memory bus. The second cache of course has some performance enhancing effects.

Chapter 5

Conclusion

The aim of this thesis was to improve the computational efficiency of lattice Boltzmann methods on complex geometries. To meet the numerical requirements of complex geometries the Bouzidi et.al. [6] bounce-back boundary conditions with interpolation were implemented into an existing LBM solver. The new boundary condition's increased arithmetic exigencies, turned out to be irrelevant, for performance. In fact the reduced amount of memory locations loaded for the boundary treatment gave the improved numerical implementation, an minor performance increase too. Thus, the enhanced LBM algorithm, the *LBM2BC* was equal without reasonable performance loss or gain in comparison to the original *LBMKernel* from Donath [8] on the empty channel test cases.

A simple test case for more complex geometries consisted of cubic channels where the incline is changed as a function over a decreasing flow opening. To put more stress onto the boundary calculation routine, an additional sphere was placed inside the channel. This altered field of application showed that the performance decreased down to 50 percent in the full matrix implementation, with a rising obstacle count. To decrease the fixed obstacle performance impact an unstructered list layout was implemented. This layout stores only computational relevant cells and diminishes the amount of memory. To compensate the greatest disadvantage, the indirect index addressing, blocking was applied to increase spatial data locality.

On the AMD Opteron based system the unstructered list layout stayed close behind the full matrix implementations. The quite same performance of full matrix and unstructured list implementation was seen on the Intel Xeon test system. Due to the excellent prefetch optimizations, applicable to full matrix layouts on the IA64 system, the unstructered list layout performs worse on empty channels. However the performance decrease of the full matrix implementation with growing complexity, was not discovered with the unstructured list layout, which stayed nearly constant over all tested domain sizes and flow openings. Dependent on the platform, the unstructured list implementation is the better performing algorithm if the amount of fluid is up between 60 to 80 percent of an cubic domain.

To address multi processor compute nodes, a simple parallelization approach was done using OpenMP. The standard approach showed problems with the IA64 compiler prefetch algorithm, which were corrected with an advanced manual scheduling. As a consequence the parallel version with one processor performed equally to the serial program version an all test systems. On both Intel based test systems, one equipped with an Intel Xeon and one with an Intel Itanium2 the performance gains of an additional processor were between 20 to 6 percent. The single shared path to the memory, equally on both systems, seems to be the bottleneck, as theoretical assumptions show that performance of the arithmetic units is

far away from being the bottleneck. The test system based on AMD Opteron processors circumvents this bottleneck with its more sophisticated memory architecture, and one path to the memory for each processor. The results support the memory bandwidth sensitivity of LBM, because a nearly ideal scaling speedup is seen.

To support more complex scenarios, than the test cases used in this thesis, such as blood vessel flow and porous media simulation, the unstructured list based algorithm seem to have the most potential and advanced boundary conditions are required for physical correctness.

Bibliography

[1] Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. http://developer.intel.com/, 2002. last visited 03.01.2006.

[2] Intel® Itanium® 2 Processor Reference Manual. http://www.intel.com/, 2004. Document Number: 251110-003.

[3] IA-32Intel® Architecture Optimization Reference Manual. http://www.intel.com/, 2005. Document Number: 248966-012.

[4] Optimizing Applications with the Intel® C++/Fortran Compiler. http://www.intel.com/, 2005.

[5] P. Bhatnagar, E. P. Gross, and M. K. Krook. A model for collision processes in gases. I. small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94(3):511–525, May 1954.

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

[7] S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Annu. Rev. Fluid Mech.*, 30:329–364, 1998.

[8] S. Donath. On optimized implementations of the lattice Boltzmann method on contemporary high performance architectures. Bachelor's thesis, Chair of System Simulation, University of Erlangen-Nuremberg, Germany, 2004.

[9] S. Donath. Private communication, 2005.

[10] S. Donath, T. Zeiser, G. Hager, J. Habich, and G. Wellein. Optimizing performance of the lattice Boltzmann method for complex structures on cache-based architectures. In *Proceedings: "Simulationstechnique" ,ASIM*, 2005.

[11] K. Iglberger. LBM-optimization. Bachelor's thesis, Chair of System Simulation, University of Erlangen-Nuremberg, Germany, 2003.

[12] D. Kandhai, A. Koponen, A. Hoekstra, M. Kataja, J. Timonen, and P. M. A. Sloot. Implementation aspects of 3D lattice-BGK: Boundaries, accuracy, and a new fast relaxation method. *J. Comput. Phys.*, 150:482–501, 1999.

[13] R. Mei, W. Shyy, D. Yu, and L.-S. Luo. Lattice Boltzmann method for 3-D flows with curved boundary. *J. Comput. Phys.*, 161(2):680–699, 2000.

[14] T. Pohl, F. Deserno, N. Thürey, U. Rüde, P. Lammers, G. Wellein, and T. Zeiser. Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. In *Proceedings of Supercomputing Conference SC04, Pittsburgh*, 2004.

[15] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhys. Lett.*, 17(6):479–484, Jan. 1992.

[16] Software Optimization Guide for AMD64 Processors. http://www.amd.com/, September 2005. Revision: 3.06.

[17] AMD Opteron Product Data Sheet. http://www.amd.com/, June 2004. Revision: 3.11.

[18] Tecplot,Inc. Tecplot 10 user's manual. http://www.tecplot.com/, March 2005.

[19] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, accepted for publication, 2004.

[20] D. A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*, volume 1725 of *Lecture Notes in Mathematics*. Springer, Berlin, 2000.

[21] G. Zschaek. On advanced lattice Boltzmann methods: analysis of different boundary conditions and evaluation of a novel hybrid thermal model. Master's thesis, Chair of Fluid Mechanics and Regionales Rechenzentrum Erlangen, University of Erlangen-Nuremberg, Germany, 2005.