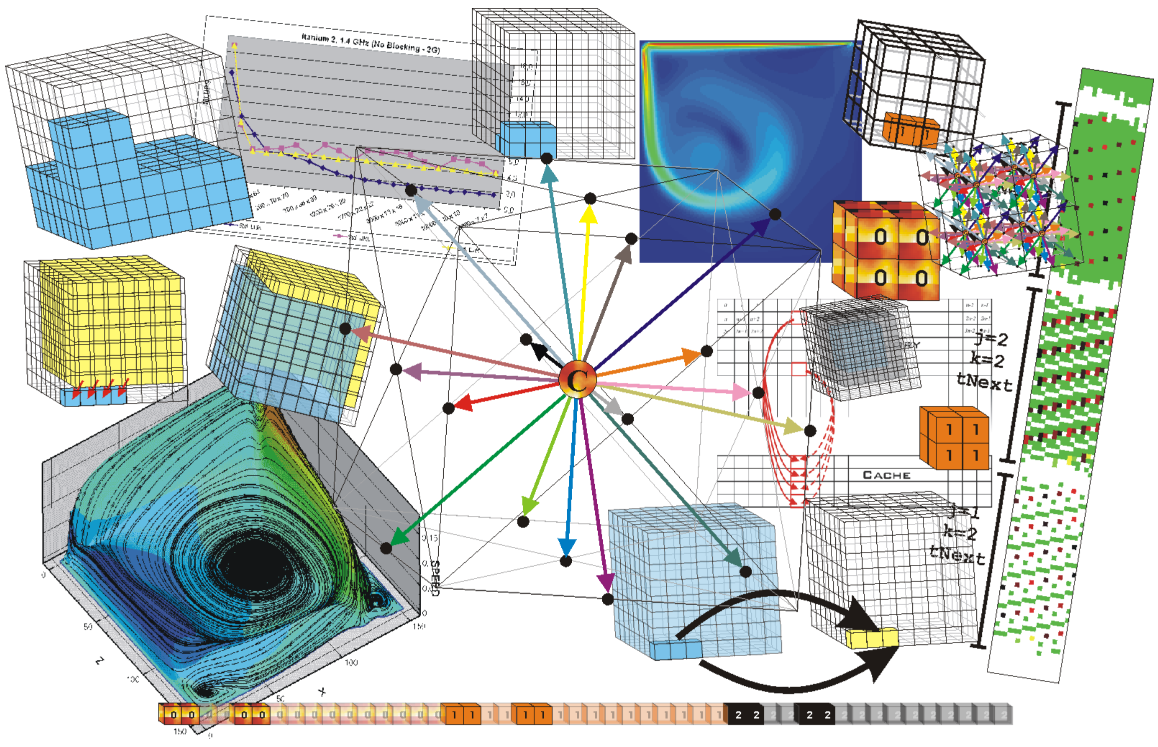
FRIEDRICH ALEXANDER-UNIVERSITÄT
ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Bachelor Thesis

On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures

Stefan Donath



On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures

Stefan Donath

Bachelor Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.-Ing. Thomas Zeiser, Dipl.-Ing. Frank Deserno, Dr. Gerhard Wellein, Dipl.-Phys. Georg Hager
Bearbeitungszeitraum:	Mai 2004 - August 2004

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 13. 8. 2004 ....

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Delivering high sustained performance for scientific applications is a well known problem in high performance computing. Especially for memory intensive applications from computational fluid dynamics, data transport from memory to processor has become the major bottleneck in particular for modern microprocessors. Introducing complex cache hierarchies is the usual hardware approach to minimize the effect of this gap. However, the efficiency of cache usage is extremely sensitive to the memory access patterns used in the application.

It is the objective of this thesis to analyze and optimize the memory access patterns in the Lattice Boltzmann method, a recent method from computational fluid dynamics. Owing to the simplicity of this method and its implementation, it easily allows the testing of different data layouts with a single, basic kernel. The performance impact of the resulting memory access patterns is analyzed on different architectures. These measurements are supplemented by theoretical investigations, like the implemented memory visualizer, profiling or the estimation of upper performance limits imposed by the hardware architectures used.

Chapter 1

Introduction

On today's architectures the performance of most scientific applications is limited by the bandwidth of the memory bus. The processor can perform operations very fast, however the main memory has high latencies and low bandwidth. Consequently, the processor has to idle while waiting for data from the main memory. To reduce this memory bottleneck, the most commonly used solution is to place a hierarchy of "caches" between the CPU and the main memory. In general a cache is a faster but smaller memory with high bandwidth, which buffers data for fast access. To achieve high performance with scientific application it is therefore necessary to make efficient use of the available cache hierarchy. Due to the limited size of the caches this means in general improving the (spatial and temporal) data locality. Depending on the numerical algorithm this can require major changes in the implementation and data layout.

It is known that the solution of computational fluid dynamics (CFD) problems is very demanding in terms of memory and CPU time. During the last decade the Lattice Boltzmann method (LBM) evolved as an alternative to classical Navier Stokes solvers for the numerical solution of fluid flows. Owing to the simplicity of the algorithm, a basic LBM flow solver ("*LBMKernel*") is used in this thesis to analyze in detail the effect of different memory access patterns on the single processor performance. For that purpose, despite using available profiling tools, also a "memory visualizer" has been developed to illustrate the memory access patterns and to ease the search for improved data layouts.

The remainder of the thesis is as follows: In the next two sections, a very brief summary of the LBM and its general implementation is given. Chapter 2 presents assumptions of the highest achievable performances on test architectures used for the *LBMKernel*, due to the limitation by the memory bandwidth. In chapter 3 basic optimization techniques and their application to the *LBMKernel* are discussed. This includes compiler switches, special tuning of loops to reduce the loop overhead and to adapt the *LBMKernel* to different architectures. Memory related optimizations are examined in chapter 4. This includes a detailed discussion of the cache hierarchy as well as different data layouts and blocking techniques. The results are presented in chapter 5. A detailed comparison of performance improvements on the test architectures is given and outcomes of profiling and memory visualizer are presented. Finally, chapter 6 summarizes the results and gives an outlook on future work.

1.1 The Lattice Boltzmann Method

The Lattice Boltzmann method is a powerful method for the numerical simulation of fluid flow. All LBM are based on the idea of lattice gas cellular automata (LGCA) to simulate the fluid motion by a simplified microscopic model in discrete time steps using a discrete phase space, i.e. discrete velocity and location. Each cell of the resulting lattice represents a volume element of the fluid, which consists of a collection of particles. Their motion is represented by a particle velocity distribution function (pdf) at each grid point. During one time step, the particles move according to the pdf into the adjacent cells (the so called propagation step) and collide with other particles streaming into this cell from different directions (the so called collision phase). The macroscopic variables of interest (i.e. density, pressure, velocity) can easily be obtained from these pdfs [WoG00]. Compared to other methods, the advantages of the LBM are fast computational speed and high accuracy, and the capability to handle complex geometries for flow-structure interactions [YML01]. From a computational point of view, the main advantage of the method is, that there are only interactions between neighboring cells, i.e. data dependencies are restricted to neighbors and allow easy implementation as well as optimization of the method.

Before going into more details of the algorithm and its implementation, a very brief summary of the physical background of LBM is given in the following. The description is mainly based on [MSY02].

Conventional CFD methods compute the flow field by numerically solving the Navier-Stokes equations themselves in space $\boldsymbol{x}$ and time t. In contrast, the Boltzmann equation deals with the single particle distribution function $f(\boldsymbol{x},\boldsymbol{\xi},t)$, where $\boldsymbol{\xi}$ denotes the particle velocity in phase space $(\boldsymbol{x},\boldsymbol{\xi})$ and time t. Through averaging the results of the microscopic/mesoscopic Boltzmann Equation can be transformed in macroscopic quantities which obey the Navier-Stokes Equations. One of the simplest kinetic models is the Boltzmann Equation with the single relaxation time approximation:

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

where $\boldsymbol{\xi}$ is the particle velocity, $f^{(0)}$ is the equilibrium distribution function (the Maxwell-Boltzmann distribution function), and λ is the relaxation time. A first step towards the LBM is the discretization of the particle velocity space using a finite set of only a few discrete velocities $\{\boldsymbol{\xi}_\alpha\}$. This can be done without severely degrading hydrodynamics,

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

In Equation 1.2, $f_\alpha(\boldsymbol{x},t) = f(\boldsymbol{x},\boldsymbol{\xi}_\alpha,t)$ is the distribution function and $f_\alpha^{(eq)}(\boldsymbol{x},t) = f^{(0)}(\boldsymbol{x},\boldsymbol{\xi}_\alpha,t)$ is the equilibrium distribution function of the α-th discrete velocity $\boldsymbol{\xi}_\alpha$, respectively. A typical set of velocities for 3D simulations is the so called D3Q19 model. It consists of 18 different directions and the center, which stands for particles with velocity zero, as can seen in Figure 1.1. Compared to other common 3D models like the D3Q15 and D3Q27 model, the D3Q19 model is a good compromise in terms of stability and computational effort [MSY02]. For athermal fluids, the equilibrium distribution for 3D models is of the form

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

where w_α is a weighting factor for the different directions, $\boldsymbol{e}_\alpha$ is a discrete velocity and $c = \delta x/\delta t$ is the so-called lattice speed (δx and δt are the cell size and the time step).
The D3Q19 model has the following set of discrete velocities (see Figure 1.1 for numbering of directions):

$$\boldsymbol{e}_\alpha = \begin{cases} (0,0,0), & \alpha = 0 \\ (\pm1,0,0)c,\ (0,\pm1,0)c,\ (0,0,\pm1)c, & \alpha = 2,4,6,8,9,14 \\ (\pm1,\pm1,0)c,\ (0,\pm1,\pm1)c,\ (\pm1,0,\pm1)c, & \alpha = 1,3,5,7,10,11,12,13,15,16,17,18 \end{cases} \tag{1.4}$$

The weighting factor w_α depends on the direction:

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

It can be shown that $f_\alpha^{(eq)}$ is a Taylor series expansion of the Maxwellian $f^{(0)}$. This approximation of $f^{(0)}$ by the above $f_\alpha^{(eq)}$ makes the method valid only in the incompressible limit $u/c \to 0$, and if $|\delta x| = |\delta t| = 1$. With the velocity space discretized, the hydrodynamic moments of f and $f^{(0)}$ are evaluated by the following quadrature formulas:

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

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

The viscosity of the fluid is $\nu = \lambda c_s^2$. Discretizing Equation 1.2 in space $\boldsymbol{x}$ and time t leads to:

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

where $\omega = \frac{\delta t}{\lambda}$. The viscosity in the Navier-Stokes equation derived from Equation 1.8 is $\nu = \frac{1}{6}(\frac{2}{\omega} - 1)$. This equation can be solved in the following two steps:

$$\text{collision step:} \quad \tilde{f}_\alpha(\boldsymbol{x}_i, t) = f_\alpha(\boldsymbol{x}_i, t) - \omega \left[f_\alpha(\boldsymbol{x}_i, t) - f_\alpha^{(eq)}(\boldsymbol{x}_i, t) \right], \tag{1.9}$$

$$\text{streaming step:} \quad f_\alpha(\boldsymbol{x}_i + \boldsymbol{e}_\alpha \delta t, t + \delta t) = \tilde{f}_\alpha(\boldsymbol{x}_i, t), \tag{1.10}$$

where $\tilde{f}_\alpha$ is the distribution function after collision. The collision step takes the collision of the particles into account and smoothes the distribution functions towards the total sum of 1 (equilibrium state).

Equations 1.9 and 1.10 implement the *collide-stream* order, which is also known as the *push method*, where first the distribution functions are read from current cell, and then the updated values are written to the adjacent cells.

1.2 The LBMKernel

It is obvious that the order of collision step and streaming step can be reversed. Instead of pushing the new values of the current cell's computation to the adjacent cells, in *stream-collide* order (also called *pull method*) the distribution functions of neighbored cells are first drawn into the current cell and then the collision step is performed there. Since the results of [Igl03] show in most cases better performance for the collide-stream method only this method is investigated in this thesis.

The basic LBM solver ("*LBMKernel*") used in this thesis is written in Fortran90 and uses the D3Q19 as described above.

Since the *LBMKernel* is written in Fortran, all algorithms printed in this thesis are in Fortran style. To show more clearly how the program works, few simplifications are made. All optimization methods that will be introduced are done by the use of macros. To keep clarity, most of them will not

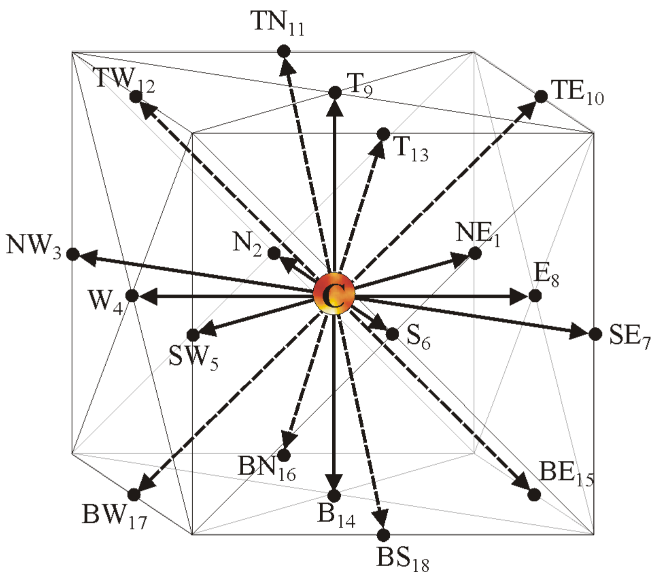


Figure 1.1: Lattice site and its directions for D3Q19 model. The directions are labeled according to compass notation while T refers to "top" and B to "bottom". The numbers denote the position in the array used by LBMKernel.

be printed in code examples. The most important macros are `IJKL_R` and `IJKL_W` macros which are used to implement different memory layouts by interchanging the indices, where `IJKL_R` is used for reading values and `IJKL_W` for writing, respectively. (For details see section 4.1).

Equations 1.9 and 1.10 (see section 1.1) describe one timestep of LBM. The *LBMKernel* merges the push operation in one subroutine. First, each cell is tested if it is an obstacle cell. If not, the actual values of its `pdf` are read, then the density and velocities are calculated. At the end, the updated values are written to the corresponding adjacent cells:

Algorithm 1.1: Skeleton for collision-subroutine

```
do k = 1, kEnd, 1
  do j = 1, jEnd, 1
    do i = 1, iEnd, 1
      If .not. obstacleField(i,j,k) then

        ! Read all particle velocities from current cell

        ! Calculate density and velocities

        ! Write results of collision to adjacent cells

      end if
    enddo
  enddo
enddo
```

For the computation of $f_\alpha^{(eq)}$ at a lattice site, ρ and $\boldsymbol{u}$ (u_x, u_y, u_z) have to be determined. ρ results from the summation of all particle velocity distribution of the current lattice site. The components of a cell's velocity are calculated from the components of the particle velocity distribution, that contribute to the respective direction (x,y,z), and ρ.

Algorithm 1.2: Computation of Rho and components of cell's velocity (unoptimized)

```
! Compute density (rho)
d_tmp = 0.0_dp

do L = 0, 18
  d_tmp = d_tmp + pdf(IJKL_R(i,j,k,L,tNow))
enddo


! Compute velocities
ux = 0.0_dp
uy = 0.0_dp
uz = 0.0_dp

do L = 0, 18
  ux = ex(L) * pdf(IJKL_R(i,j,k,L,tNow)
  uy = ey(L) * pdf(IJKL_R(i,j,k,L,tNow)
  uz = ez(L) * pdf(IJKL_R(i,j,k,L,tNow)
enddo

ux = ux / d_tmp
uy = uy / d_tmp
uz = uz / d_tmp
```

Collision and streaming are performed in one step. The following example shows a nearly completely unoptimized version, which uses a straightforward implementation of Equation 1.11, a representation of Equation 1.9 which saves some arithmetic.

$$\tilde{f}_\alpha(\boldsymbol{x}_i,t) = (1-\omega) f_\alpha(\boldsymbol{x}_i,t) + \omega f_\alpha^{(0)}(\boldsymbol{x}_i,t). \tag{1.11}$$

The results are stored in the corresponding direction of the adjacent cell. Using two different grids for data these adjacent cells are located in the target lattice, denoted by `tNext` in `pdf` array (Algorithm 1.3).

Algorithm 1.3: Collision and Streaming into adjacent cells (unoptimized)

```
! Push new values in adjacent cells
do L = 0, 18
  pdf(IJKL_W(i+ex(L), j+ey(L), k+ez(L), L, tNext)) =      &
      (1.0_dp - omega) * pdf(IJKL_R(i, j, k, L, tNow)) + &
      omega * w(L) * d_tmp * (1.0_dp + (ex(L)*ux+ey(L)*uy+ez(L)*uz) + &
      (9.0_dp/2.0_dp)*(ex(L)*ux+ey(L)*uy+ez(L)*uz)*(ex(L)*ux+ey(L)*uy+ez(L)*uz) - &
      (3.0_dp/2.0_dp)*(ux*ux+uy*uy+uz*uz))
enddo
```

The above implementation of the stream step pays no attention to obstacle cells when writing back. If the current cell is neighboring an obstacle region, the new components of `pdf` into the direction of the obstacles are written blindly into the corresponding direction of the obstacle cell. Since that cannot happen in reality, it would be necessary to test for each direction, if the adjacent cell is an obstacle. The consequence is an additional 'if' statement in write back loop. That would lead to a significant drop in performance. Therefore, these checks are done in an additional subroutine, called "bounce back" (see Algorithm 1.4). Its task is to write the `pdf` values that were mistakenly written to an obstacle cell back into opposite direction of the originating cell.

To avoid complications when handling boundary conditions, the grid contains two more cells in each direction than are actually needed. These so called *ghost layers* lie around the object. Figure 1.2

Algorithm 1.4: Bounce back routine for obstacle cells (unoptimized)

```
do k=1,kEnd
   do j=1,jEnd
      do i=1,iEnd
         if ( obstacleField(i,j,k) ) then
            do L = 1, 18
               pdf(IJKL_R(i+ex(L),j+ey(L),k+ez(L),L,tNow) = pdf(IJKL_R(i,j,k,L_op(L),tNow)
            enddo
         endif
      enddo
   enddo
enddo
```

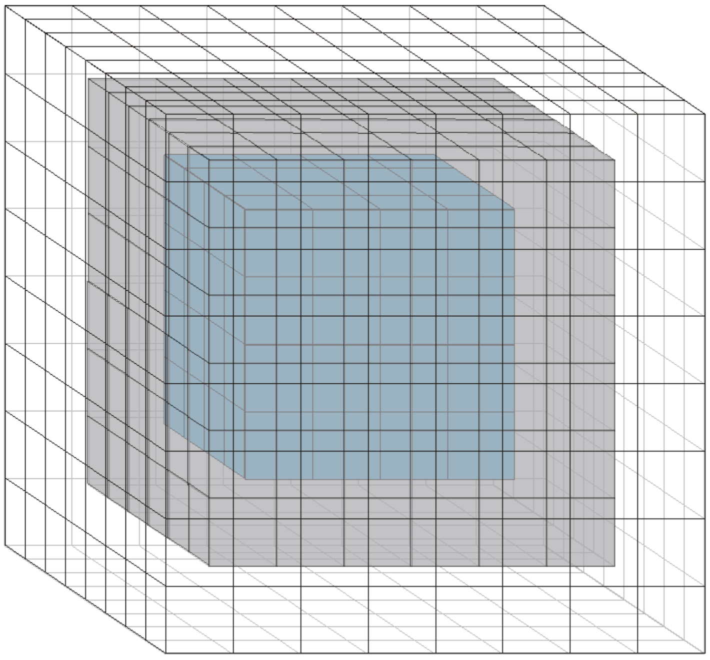


Figure 1.2: Model's fluid (blue) is surrounded by the boundary obstacles (grey). The grid used in LBMKernel contains one additional complete enclosing ghost layer.

illustrates a cube of fluid surrounded by the obstacle cells representing wall boundaries, which are enclosed themselves by the ghost layers.

1.3 Lid Driven Cavity

During optimization of codes it is often necessary to reorder instructions or to change the algorithm. Thus, it is very important to verify that the implementation is still calculating correctly. Since the task is not to find the best way of simulation for fluid flows, but to find techniques for code optimization, a very simple application of LBM is used: The *Lid Driven Cavity* describes the motion of fluid in a box where the top wall is endlessly sliding into same direction at constant speed. This is simulated by assigning the cells of the first layer a constant speed in x-direction. To prevent any side-effects near the borders, not all the cells of this layer are accelerated. Five cells at each border will be simulated by the collision routine as supplied before (see Algorithm 1.5).

Algorithm 1.5: If statement for Lid Driven Cavity

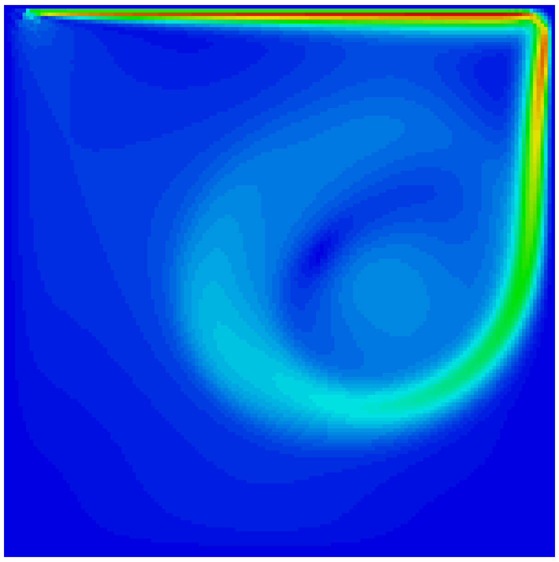
```
#ifdef LIDDRIVENCAVITY
  if ((k .eq. 2).and.(i .gt. 7).and.(i .lt.(iEnd-7)).and.(j .gt. 7).and.(j .lt.(jEnd-7)))
      then
    ux    = acc
    uy    = 0.0_dp
    uz    = 0.0_dp
    d_tmp = 1.0_dp
  end if
#endif
```

To test and verify that the program computes correctly, the acceleration of top layer cells can be built in by using a macro. The statements which are added this way could be easily optimized if they were inherent parts of the code. There are several techniques to eliminate such 'if' statements as shown in Algorithm 1.5. But if that were done, some of the optimizations presented here were not portable to other Lattice Boltzmann problems, or even to other numerical codes, any more, because they were too specific for the Lid Driven Cavity.

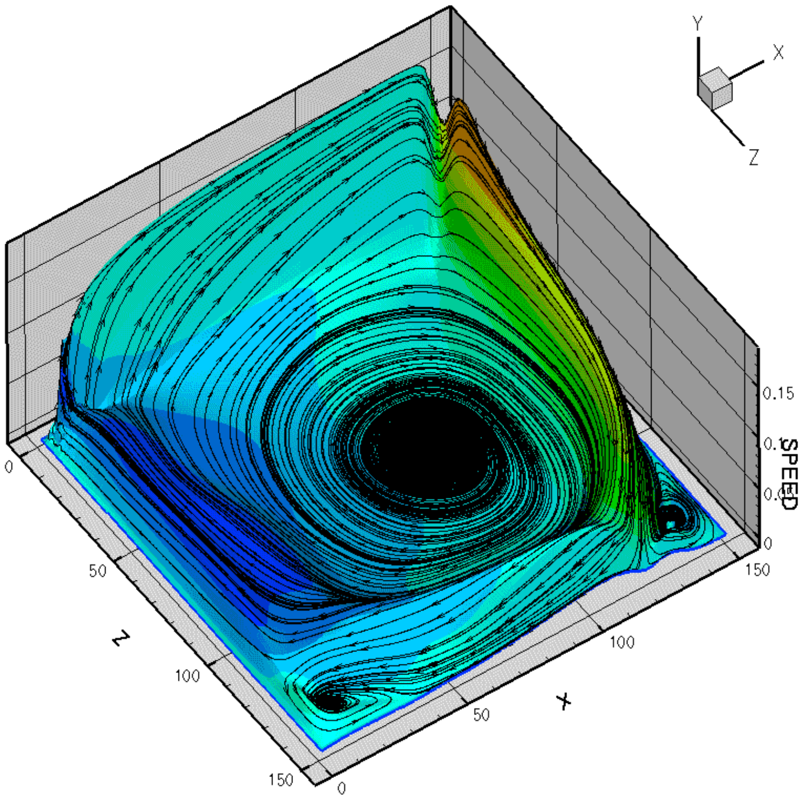
For verification the *LBMKernel* was expanded by routines that write out one layer of the grid, or the whole three dimensional field, optionally. For visualization the output files can be read in by programs like "TecPlot"[1] or "LatticeView"[2]. For examples see Figure 1.3.

[1]TecPlot is a program to plot and animate simulations and experimental data, developed by Amtec Engineering, Inc., `http://www.amtec.com`

[2]LatticeView, a tiny program that creates an image of a two dimensional lattice, where the color of each pixel represents some value of the corresponding cell. Programmed by F. Deserno, IMMD10, University of Erlangen-Nuremberg, `http://www10.informatik.uni-erlangen.de/~deserno/`



(a) LatticeView image showing the fluid flow in the 60th xz-layer of a 150^3 field.



(b) Two dimensional layer grabbed out of a 150^3 grid. The arrowed lines show the flow of fluid, y-axes and color denote the speed. Produced by TecPlot.

Figure 1.3: Visualization of Lid Driven Cavity flow. The sliding lid moves the fluid in a circular motion.

Chapter 2
Performance Limiting Aspects

When optimizing codes one of the most interesting facts is the ratio of the current to the maximal achievable performance. The maximal achievable performance is determined by several limiting factors. These include the theoretical peak performance and the memory interface. Depending on the algorithm, the CPU might not deliver peak performance since data dependencies or the order of instructions can cause some arithmetic units of processor to be temporary idle. In particular for codes which operate on large data sets, the size and organization of the caches as well as the Translation Lookaside Buffers (TLBs) are of major importance. On distributed shared memory systems (DSM), also the bandwidth of buses that interconnect the single nodes and the additional communication overhead can be a limiting factor.

Since the collision routine of *LBMKernel* has a memory operations per floating point operations ratio which is relatively high and constant when scaling the problem size, it can be regarded as memory intensive, and thus, the main performance limiting aspect for *LBMKernel* is the memory bandwidth of the architecture used. (On shared memory systems, when system size exceeds local memory the bandwidth of interconnecting buses between the nodes become main bottleneck.)

2.1 Memory Bandwidth as Performance Limiting Aspect

Although the *LBMKernel* is computational very intensive, neither the number of registers nor the number of arithmetic units on the processor will be such a limiting factor like the memory bandwidth.

For estimating the maximal achievable performance, first it is necessary to determine the amount of data that is transferred over the memory bus. Regarding the best possible case of minimal loads and stores, for each cell all 19 `doubles` which represent the values of `pdf` have to be loaded at least once. After calculating the new values, they have to be written back at least once again. For this, it is important, which write strategy the cache uses. Since in comparison to the main memory, the caches' latencies are very low and their bandwidths very high, only the write strategy of the highest level cache has to be considered. While on the architectures included in this thesis the highest level cache is organized by write-back strategy, there are also architectures that use write-through caches. So, storing to a write-through cache results always in a store to the main memory, while stores to a write-back cache results in writing of the cache line to memory not before the cache line has to be replaced (The memory store occurs when the next read capacity miss or read conflict miss occurs). When a write miss occurs, and the cache is a write-back cache, it first has to load the corresponding data from memory, so that the processor is able to store the data into this cache line (write-allocate). If the cache is write-through, there are two possibilities: Some architectures perform a write-allocate, others do not. This write-allocate is advantageous if the program has to read from data that is close to written data (locality) or has to write to the same position where it is reading from, because then this data is already present in cache. Otherwise, a cache with write-no-allocate needs no additional load when data is stored ([Pra01][AMD99]). Some of today's architectures let choose the write miss policy by directives in the program.

It is doubtful, whether an estimation for the maximum of the achievable performance shall include the probability of cache write misses. Of course, there will be no program, that operates on big amounts of data, without any cache misses. In addition, the real behavior depends on the algorithm. Considering an algorithm, that never writes to positions where it reads from (e.g. vector triad [Sch00]), every write operation will cause a write miss, so in case of an architecture with write-allocate cache every write operation will result in one load and one store to memory. In the case of the LBM, the data is dependent on itself. Algorithm 1.3 shows, that the updated values of the current cell are written to other cells, which are – in this code version – placed in an other grid (for other code versions see section 4.2), which resides on other positions of memory. So nearly every write operation will probably cause a cache write miss. Assuming now the worst case, it has to be kept in mind that the cache has to load a whole cache line from memory, when balancing the write-allocate. Thus, it has to load, depending on the length of the cache line, more data than the processor is writing, which needs more memory bandwidth.

In the end, there are three models that could be considered: First, a combination of code and architecture that is able to write without write-allocate, either by avoiding cache write misses or reasoned by architecture properties (*Model 1*). *Model 2* assumes one additional read access per write access due to write-allocate, and *Model 3* includes the consideration that a whole cache line has to be loaded when write-allocate occurs.

Thus, Model 1 regards the maximal achievable performance, if the code is completely vectorized without having write-allocate performing caches. The limit of Model 2 applies to an architecture with write-allocate caches and a code which reaches full cache reuse. The third model assumes full cache reuse for read but no reuse for written data.

Assuming that there are 19 `doubles` to load (the directions of `pdf`), and, after computation of the new values, to store back. That results in 38 memory operations per cell. Since one of these memory operations loads or stores one `double`, which is 8 Bytes, one lattice site update results in $19 \cdot 2 \cdot 8$ Bytes $= 304$ Bytes that have to be loaded or stored. On an architecture with write-allocate and write-through cache, there is in case of a write miss for each store one additional load in Model 2, and n additional loads in Model 3, where n is the length of one cache line in `doubles`. That results in 456 Bytes per lattice site update that were loaded or stored in second model, and $19 \cdot (2+n) \cdot 8$ Bytes in the third model. The probability of write misses in a write-back cache could be less, because a missed and loaded cache line could prevent a write miss for next write operation. But this depends on the order of accessing the data (see section 4.1).

The maximum of achievable performance, limited by memory bandwidth, is given by:

$$\text{maximum performance} = \frac{\text{memory bandwidth in Bytes/s}}{\text{memory accesses per cell in Bytes/cell}} \cdot 10^{-6} \tag{2.1}$$

To show, that memory bandwidth is in fact the main limiting aspect concerning performance of *LBMKernel*, the maximum of achievable performance limited by processor's peak performance is calculated, too. As collision routine performs theoretically 156 floating point operations per lattice site update, the maximum performance is given by:

$$\text{maximum performance} = \frac{\text{processor's peak performance in FLOP/s}}{156\ \text{FLOP/cell}} \cdot 10^{-6} \qquad (2.2)$$

The performance is measured in MLUPS, which stands for Mega Lattice site Updates per second. Table 2.1 shows the results for three test platforms (for details on cache line lengths see Appendix E). Due to the magnitude of difference between the memory limited and processor limited maximum of performance, it can be stated that *LBMKernel*'s performance is limited by memory bandwidth mainly.

Table 2.1: Theoretical maximum of achievable performance limited by memory bandwidth for three different models making different assumptions of architecture and code, and processor's peak performance. While Model 1 is based on highly optimistic assumptions and Model 2 does not include all important real aspects, Model 3 estimates very pessimistically. Values are of unit MLUPS.

architecture	Platform 1	Platform 2	Platform 3
Model 1	14.0	21.1	21.1
Model 2	9.4	14.0	14.0
Model 3	1.6	2.3	2.3
Proc's peak perf.	34.0	38.5	35.9

Chapter 3

Basic Optimizations

Before significant effort is spent in optimizing a code for best operation with respect to the problem of memory-bottleneck, there are some possibilities to gain more performance by using other techniques. Since most numerically codes are highly computational intensive, it is reasonable to minimize the number of floating point operations as far as the algorithm and exactness permit. In that way, architectural properties can be used, and some arithmetic can be saved. To increase the floating point operations per second ratio and the instructions per cycle ratio there are techniques of reordering instructions (or even micro-ops, the tiny operations one instruction is divided in by the processor) can be applied. Since these optimizations are closely connected to the attributes of the architecture, they have to be done by the compiler. Because of that it is important to use the best available compiler for each architecture.

3.1 Arithmetic Optimizations

Regarding Algorithms 1.2 and 1.3 in section 1.2 which show the computation of the non-equilibrium distribution and collision (see Equation 1.11), it is obvious that computations can be saved there. By scalar replacement and redundancy elimination the number of floating point operations was reduced. Also loop invariant code motion was done by hand, divisions were saved by using the multiplicative inverse and the weighting factors w_α are realized as constants. Although the compiler should perform these optimizations, too, the manual work resulted in a strong increase of performance. Comparing the unoptimized Algorithms with Algorithm D.1 in Appendix D, it is shown that there are still further optimizations done, which compiler cannot do: Since the components of discrete velocity (`ex`, `ey`, `ez`) contain many zero values, multiplications that are done in vain can be saved by unrolling the l-loops manually and removing the unnecessary computations. While now there are more temporary variables, the number of floating point operations could be reduced to 156 per lattice site update.

3.2 Logical Optimizations

Following the idea to save memory access operations, it is wise to have a look at the bounce back routine, too. The standard implementation as shown in Algorithm 1.4 traverses the whole grid in every time step. If there are few obstacle cells, as it is the case when applying Lid Driven Cavity, most of the loop iterations are done in vain. Since a conditional statement can cause long pipeline stalls if the processor's branch prediction decides wrong, avoiding the 'if' statement will increase the overall performance. For this it has to be known where all the obstacles are. As *LBMKernel* should not only be applicable to standard Lid Driven Cavity, where only the boundaries of the field are obstacles, but also more complex constructs with obstacles inside, it is not sufficient to perform the bounce back on boundaries only. For this reason, there is another solution: Before beginning the first LBM step, a list of all obstacles is generated. Their positions are saved in an extra array. Algorithm 3.1 shows an excerpt of the routine `make_bouncebacklist`, which examines the `obstacleField` and writes obstacle information into `barrierIdx`. The number of obstacle cells is stored in `barrierCnt`. Of course, only those obstacles are stored, which have at least one adjacent fluid cell (this is assured by the complex if-statement).

Algorithm 3.1: Routine make_bouncebacklist creates a list of all obstacles

```
barrierCnt = 0

! Examine all cells of lattice
do k=1,kEnd
    kp = min(k+1,kEnd)
    km = max(k-1,1)
    do j=1,jEnd
        jp = min(j+1,jEnd)
        jm = max(j-1,1)
        do i=1,iEnd
            ip = min(i+1,iEnd)
            im = max(i-1,1)

            ! Is cell an obstacle which is not enclosed by other obstacles?
            if ( obstacleField(i  ,j  ,k  ) .and. .not. &
                ( &
                  obstacleField(im,jm,k  ) .and.   &
                  obstacleField(i  ,jm,k  ) .and.   &
                  obstacleField(ip,jm,k  ) .and.   &
                  obstacleField(im,jp,k  ) .and.   &
                  obstacleField(i  ,jp,k  ) .and.   &
                  obstacleField(ip,jp,k  ) .and.   &
                  obstacleField(ip,j  ,k  ) .and.   &
                  obstacleField(im,j  ,k  ) .and.   &
                  obstacleField(im,j  ,km) .and.   &
                  obstacleField(i  ,j  ,km) .and.   &
                  obstacleField(ip,j  ,km) .and.   &
                  obstacleField(i  ,jm,km) .and.   &
                  obstacleField(i  ,jp,km) .and.   &
                  obstacleField(im,j  ,kp) .and.   &
                  obstacleField(i  ,j  ,kp) .and.   &
                  obstacleField(ip,j  ,kp) .and.   &
                  obstacleField(i  ,jm,kp) .and.   &
                  obstacleField(i  ,jp,kp) ) &
                ) then
                barrierCnt = barrierCnt + 1
                barrierIdx(1,barrierCnt) = i
                barrierIdx(2,barrierCnt) = j
                barrierIdx(3,barrierCnt) = k
            endif
        enddo
    enddo
end do
```

The bounce back routine can now use the information listed in `barrierIdx`. Querying whether the particle velocities of current cell has to be written back into the originating cell is not necessary any more, because now the assignments are performed for the stored cells in `barrierIdx`. The optimized routine (see Algorithm 3.2), which is called every time step, handles the bounce back much faster, despite the indirect addressing.

Algorithm 3.2: Bounce back routine traverses list of obstacles

```
! perform bounce back for all cells known as barrier
do m=1,barrierCnt

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

   pdf(IJKL_R(i+1,j+1,k  ,1 ,tNow))  = pdf(IJKL_R(i,j,k,5 ,tNow))
   pdf(IJKL_R(i  ,j+1,k  ,2 ,tNow))  = pdf(IJKL_R(i,j,k,6 ,tNow))
   pdf(IJKL_R(i-1,j+1,k  ,3 ,tNow))  = pdf(IJKL_R(i,j,k,7 ,tNow))
   pdf(IJKL_R(i-1,j  ,k  ,4 ,tNow))  = pdf(IJKL_R(i,j,k,8 ,tNow))
   pdf(IJKL_R(i-1,j-1,k  ,5 ,tNow))  = pdf(IJKL_R(i,j,k,1 ,tNow))
   pdf(IJKL_R(i  ,j-1,k  ,6 ,tNow))  = pdf(IJKL_R(i,j,k,2 ,tNow))
   pdf(IJKL_R(i+1,j-1,k  ,7 ,tNow))  = pdf(IJKL_R(i,j,k,3 ,tNow))
   pdf(IJKL_R(i+1,j  ,k  ,8 ,tNow))  = pdf(IJKL_R(i,j,k,4 ,tNow))
   pdf(IJKL_R(i  ,j  ,k+1,9 ,tNow))  = pdf(IJKL_R(i,j,k,14,tNow))
   pdf(IJKL_R(i+1,j  ,k+1,10,tNow))  = pdf(IJKL_R(i,j,k,17,tNow))
   pdf(IJKL_R(i  ,j+1,k+1,11,tNow))  = pdf(IJKL_R(i,j,k,18,tNow))
   pdf(IJKL_R(i-1,j  ,k+1,12,tNow))  = pdf(IJKL_R(i,j,k,15,tNow))
   pdf(IJKL_R(i  ,j-1,k+1,13,tNow))  = pdf(IJKL_R(i,j,k,16,tNow))
   pdf(IJKL_R(i  ,j  ,k-1,14,tNow))  = pdf(IJKL_R(i,j,k,9 ,tNow))
   pdf(IJKL_R(i+1,j  ,k-1,15,tNow))  = pdf(IJKL_R(i,j,k,12,tNow))
   pdf(IJKL_R(i  ,j+1,k-1,16,tNow))  = pdf(IJKL_R(i,j,k,13,tNow))
   pdf(IJKL_R(i-1,j  ,k-1,17,tNow))  = pdf(IJKL_R(i,j,k,10,tNow))
   pdf(IJKL_R(i  ,j-1,k-1,18,tNow))  = pdf(IJKL_R(i,j,k,11,tNow))
enddo
```

3.3 Architecture Specific Optimizations

In addition to the standard version of the collision routine, as implemented in Algorithm D.1, two extra versions have been developed, which should be more adequate for different architectures. When a code performs the same operation on different data many times, it is useful to think about writing it in a way that can be vectorized. Also, a special optimization for RISC processors can be appropriate.

3.3.1 Vector Version

Vector processors execute arithmetic operations on a large number of vector elements at the same time. They are therefore optimized for huge loops with many floating point operations, which show no data dependencies within the loops. Vector processors are characterized by being optimized for floating point operations (in contrast to scalar processors, which are designed as general-purpose CPUs), faster memory access times (resulting from a more sophisticated memory architecture with several thousand banks), deep pipelines and few but big vector registers. These vector registers store not only one but several values (typically 64, 128 or 256). That group of values is called a vector, which can be a part of a multi-dimensional array. On a vector machine, a very long loop is chopped into several pieces, each of the length of a vector register and one remainder with a shorter vector. The values in a vector register are computed the way the code describes, by chaining the needed arithmetic units on the processor. This way a vector processor performs whole operation (that can be a vector addition or even more complex arithmetic) for one vector element in few cycles. Due to the long pipelines vector processors have a big overhead for short loops. The fast memory access times are realized by an expensive memory bus architecture which provides high bandwidth. Similar as on scalar architectures, the technique of interleaving results in higher bandwidth and hiding the latency of a single bank. On systems with caches stride-1-access clearly delivers the best memory performance. As vector machines usually load the data directly from main memory to the vector registers, there is no severe performance degradation for non-unit (but constant) stride as long as the access does not result in bank conflicts, which are similar costly as on scalar architectures. Several ideas that were realized in vector architectures, such as pipelining and bypassing, can nowadays also be found in superscalar processors. By using prefetching on scalar architectures, a similar effect as latency hiding can be obtained. This is why vector optimizations often show good results on scalar architectures, too [GHo01].

Algorithm 3.3: Fragments of collision routine optimized for vector architectures

```
subroutine col_optvec_??(pdfNow,pdfNext,omega,iEnd,jEnd,kEnd,obstacleField,null,rho,acc)
#define IJKL ...
! declaration of variables and constants
  omega_2 = 2.0_dp * omega
  ImOmega = 1.0_dp - omega

! update whole Grid one time
  do m=0, ((iEnd+2)*(jEnd+2)*(kEnd+2))-1
    if ( .not. obstacleField(m,0,0) ) then
      ! Read particle velocity distribution from current cell
      d1    = pdfNow(IJKL_R(m,0,0, 1)) + pdfNow(IJKL_R(m,0,0, 7)) + &
              pdfNow(IJKL_R(m,0,0, 8))
      d2    = pdfNow(IJKL_R(m,0,0, 3)) + pdfNow(IJKL_R(m,0,0, 4)) + &
              pdfNow(IJKL_R(m,0,0, 5))
      d3    = ...
      d4    = ...
      d_tmp = pdfNow(IJKL_R(m,0,0, 0)) + pdfNow(IJKL_R(m,0,0, 2)) + &
              pdfNow(IJKL_R(m,0,0, 6)) + d1 + d2+ d3 + d4
      id_tmp = 1.0_dp / d_tmp

      ! Compute velocities
      ...  ! ( equivalent to standard implementation )

      ! Compute non-equilibrium distribution
      ...  ! ( equivalent to standard implementation )

      ! Push new values in adjacent cells
       pdfNext(IJKL_R(m   ,0     ,0     , 0)) = pdfNow(IJKL_R(m,0,0, 0)) * ImOmega + ne0
       pdfNext(IJKL_R(m+1 ,1     ,0     , 1)) = pdfNow(IJKL_R(m,0,0, 1)) * ImOmega + ne1
       pdfNext(IJKL_R(m   ,1     ,0     , 2)) = pdfNow(IJKL_R(m,0,0, 2)) * ImOmega + ne2
       pdfNext(IJKL_R(m-1 ,1     ,0     , 3)) = pdfNow(IJKL_R(m,0,0, 3)) * ImOmega + ne3
       pdfNext(IJKL_R(m-1 ,0     ,0     , 4)) = pdfNow(IJKL_R(m,0,0, 4)) * ImOmega + ne4
       pdfNext(IJKL_R(m-1 ,null-1,0     , 5)) = pdfNow(IJKL_R(m,0,0, 5)) * ImOmega + ne5
       pdfNext(IJKL_R(m   ,null-1,0     , 6)) = pdfNow(IJKL_R(m,0,0, 6)) * ImOmega + ne6
       pdfNext(IJKL_R(m+1 ,null-1,0     , 7)) = pdfNow(IJKL_R(m,0,0, 7)) * ImOmega + ne7
       pdfNext(IJKL_R(m+1 ,0     ,0     , 8)) = pdfNow(IJKL_R(m,0,0, 8)) * ImOmega + ne8
       pdfNext(IJKL_R(m   ,0     ,1     , 9)) = pdfNow(IJKL_R(m,0,0, 9)) * ImOmega + ne9
       pdfNext(IJKL_R(m+1 ,0     ,1     ,10)) = pdfNow(IJKL_R(m,0,0,10)) * ImOmega + ne10
       pdfNext(IJKL_R(m   ,1     ,1     ,11)) = pdfNow(IJKL_R(m,0,0,11)) * ImOmega + ne11
       pdfNext(IJKL_R(m-1 ,0     ,1     ,12)) = pdfNow(IJKL_R(m,0,0,12)) * ImOmega + ne12
       pdfNext(IJKL_R(m   ,null-1,1     ,13)) = pdfNow(IJKL_R(m,0,0,13)) * ImOmega + ne13
       pdfNext(IJKL_R(m   ,0     ,null-1,14)) = pdfNow(IJKL_R(m,0,0,14)) * ImOmega + ne14
       pdfNext(IJKL_R(m+1 ,0     ,null-1,15)) = pdfNow(IJKL_R(m,0,0,15)) * ImOmega + ne15
       pdfNext(IJKL_R(m   ,1     ,null-1,16)) = pdfNow(IJKL_R(m,0,0,16)) * ImOmega + ne16
       pdfNext(IJKL_R(m-1 ,0     ,null-1,17)) = pdfNow(IJKL_R(m,0,0,17)) * ImOmega + ne17
       pdfNext(IJKL_R(m   ,null-1,null-1,18)) = pdfNow(IJKL_R(m,0,0,18)) * ImOmega + ne18
    end if
  enddo
end subroutine
```

To adapt the *LBMKernel* better to vector architectures, the most important issue that has to be changed are the three nested loops of the collision routine which are short in terms of vector length (see skeleton in Algorithm 1.1). As mentioned above, a vector computer works better the longer the loop is. Thus, in the vector optimized version the three loops running over the whole lattice are merged into one big loop, which runs from 0 to the last lattice site. This requires that the three spatial indices follow one after another in array declaration (Therefore Vector version cannot be applied to all data layouts discussed in section 4.1). This one loop runs also over the ghost layer cells which lie outside of obstacle boundaries (see Figure 1.2). Again, the ghost layers show the reason of their existence. If there were no ghost layers, the extra boundary conditions would make a merge of the loops impossible. See the indications in Algorithm 3.3 to compare with the Standard implementation (Algortihm D.1). Two more aspects should be mentioned: The subroutine for vector collision receives two parameters for `pdf` instead of one. The calling routine passes the array `pdf(0,0,0,0,tNow)` into `pdfNow`, and `pdf(0,0,0,0,tNext)` into `pdfNext`. The reason is to give the compiler the hint to store the two parts of the array in two different vector registers. The way of addressing the data in `pdfNow` and `pdfNext` by the use of one running index m formally violates the array boundaries. Although the Fortran standard allows a redeclaration of array boundaries, so that the arrays could be handled as two dimensional, they are kept as four dimensional due to reasons of uniformity. Therefore, it is necessary to use negative values for indexes to address the adjacent cells in write back statements. As the compiler might report an error, because negative indexes are not allowed by the array declaration used, the negative values are produced by subtracting them from the parameter `null` which receives the value zero from calling routine.

3.3.2 RISC Version

There were also efforts to find an optimized version for RISC processors. The idea is, to gain small amounts of work, that can be easily handled by the arithmetic units on the processor without lag due to data dependencies and register shortages, and to reach better cache reuse. Algorithm 3.4 shows fragments of RISC version of *LBMKernel*. The collision step that is handled separated for each cell exists no longer. Instead, the collision step is partitioned in several small pieces, that are performed for the whole dimension i. This gives the compiler the possibility to handle with loop unrolling, fusing and other standard techniques to get an optimal utilization of arithmetic units and registers. Note primarily the 19 separated loops for writing the results to adjacent cells. The main aspect of the RISC version is the possibility of getting stride-1-access over one dimension of the lattice by adapting the memory layout. When using the index i as first dimension in array (which is in Fortran data layout that one which addresses adjacent memory locations), all the small loops running over i can gain full cache data reuse (for details on memory layouts see section 4.1). Above all, the write back operation into the adjacent cells is improved that way. While writing to memory locations that are far away from the current cell (in memory) in Standard version causes many cache write misses, the 19 separated write loops perform this operation with stride-1-access, thus, with full cache reuse after a cache write miss. The drawback of RISC version is: for storing the intermediate results for non-equilibrium state, velocities and density, this algorithm needs 25 additional arrays of the length of `iEnd` instead of simple variables (`ne0`..`ne18`, `d1`..`d4`, `d_tmp` and `id_tmp`). That means additional memory usage, and, of course, additional cache usage. In third loop – the biggest one – there are load and/or store operations on 39 of these i-dimensional fields (`ne0`..`ne18`, `d_tmp`, `id_tmp`, `d1`..`d4`, `pdf(1)`..`pdf(3)`, `pdf(5)`..`pdf(7)`, `pdf(10)`..`pdf(13)`, `pdf(15)`..`pdf(18)`). These temporary fields can grow with system size such, that cache will not be large enough to hold enough `pdf` data and temporary fields. In chapter 5 can be seen, that RISC version suffers from this drawback on platforms with small caches.

3.4 Compiler Optimization

Compilers translate the program in a high level programming language to the executable architecture specific code. Since they are able to restructure the original code significantly and can do architecture specific optimizations they are an important factor for application performance. However, they are black boxes, which work can only be influenced with switches and by using directives in the code.

Algorithm 3.4: Schematic overview of collision routine optimized for RISC architectures

```
  real dimension(1:iEnd,0:18)  :: ne
  real dimension(1:iEnd)       :: d1,d2,d3,d4,d_tmp,id_tmp
! declaration of variables and constants

! update whole Grid one time
  do k = 1, kEnd, 1
    do j = 1, jEnd, 1
      do i = 1, iEnd, 1
        ! Read particle velocity distribution from current cell
        d1(i)    = pdf(IJKL_R(i,j,k, 1,tNow)) + pdf(IJKL_R(i,j,k, 7,tNow)) + &
                   pdf(IJKL_R(i,j,k, 8,tNow))
        d2(i)    = pdf(IJKL_R(i,j,k, 3,tNow)) + pdf(IJKL_R(i,j,k, 4,tNow)) + &
                   pdf(IJKL_R(i,j,k, 5,tNow))
        d3(i)    = ...
        d4(i)    = ...
        d_tmp(i) = pdf(IJKL_R(i,j,k, 0,tNow)) + pdf(IJKL_R(i,j,k, 2,tNow)) + &
                   pdf(IJKL_R(i,j,k, 6,tNow)) + d1 + d2 + d3 + d4
      enddo

      do i = 1, iEnd, 1
        id_tmp(i) = 1.0_dp / d_tmp(i)
      enddo

      do i = 1, iEnd, 1
        ! Compute velocities
        ...
        ! Compute non-equilibrium distribution
        ne(i,0) = d_tmp(i) * (frac1_3 - 0.5_dp * usq) * omega
        ne(i,1) = coeff_1 * ui1 * (frac2_3 + ui1 ) + usqn1
        ne(i,2) = ...
        ...
        ne(i,17) = ...
        ne(i,18) = ne(i,11) + coeff_2 * ui11
      enddo

      ! Push new values in adjacent cells
      do i = 1, iEnd, 1
        if ( .not. obstacleField(i,j,k)) then
          pdf(IJKL_W(i,j,k,0,tNext))=pdf(IJKL_R(i,j,k,0,tNow)) * ImOmega + ne(i,0)
        end if
      enddo

      do i = 1, iEnd, 1
        if ( .not. obstacleField(i,j,k)) then
          pdf(IJKL_W(i+1,j+1,k,1,tNext))=pdf(IJKL_R(i,j,k,1,tNow)) * ImOmega + ne(i,1)
        end if
      enddo

      ...

      do i = 1, iEnd, 1
        if ( .not. obstacleField(i,j,k)) then
        pdf(IJKL_W(i,j-1,k-1,18,tNext))=pdf(IJKL_R(i,j,k,18,tNow)) * ImOmega + ne(i,18)
        end if
      enddo
    enddo
  enddo
```

This section discusses the impact of compiler switches and directives on the performance.

For the test systems in this thesis (see Appendix E) following different compilers were used: Intel's Fortran compiler for IA32 architectures in version 7.1 and 8.0, Intel's Fortran compiler for IA64 architectures in version 7.1 and 8.0 and Fortran PGI compiler in version 5.0-2 and 5.1-3 from Portland Group (for details on version and build number see Appendix F).

3.4.1 Optimization by Compiler-Flags

Compiler-Flag –O3

The simplest idea for optimizing floating point operation intensive codes is, of course, switching compiler optimization to highest level, that is, for Intel's compilers `-O3` and for PGI compiler `-O4`. Intel compilers perform additionally to the standard optimizations like loop unrolling, global instruction scheduling and partial redundancy elimination, more aggressive techniques such as prefetching, loop transformations and scalar replacement.

Compiler-Flags –tpp7 and –xW

Specifying the target architecture (in case of Pentium 4 the flags `-tpp7 -xW` are used), enables the compiler to produce code that is optimized for its characteristic attributes, such as optimization for SSE2 on Pentium 4, for example. The flag `-xW` enables the compiler to use specific instructions for vectorization, which are possible only on Pentium 4 architectures.

Compiler-Flag –fno-alias

Also the flag `-fno-alias` promises an enhancement in performance. It tells the compiler that in the program no aliasing is used. Data, e.g. two arrays with different names, are aliased if they refer to identical memory locations, because a calling routine has passed same variable to two different parameters. When compiler has to expect that data can be aliased, local instruction scheduling cannot performed optimal because of data dependencies. Otherwise better utilization of processor resources can be reached [GHo01]. While Fortran77 standard forbids use of aliasing, the Fortran90 standard permits it. Intel's compilers assume aliasing by default.

3.4.2 Optimization by Directives

In addition to optimization flags, there are directives that provide hints for the compiler how to handle parts of the code. *LBMKernel* uses directives for vectorization of loops in RISC-Version, where they are placed directly ahead of the inner i-loops. The directives used are "`!DEC$ IVDEP`", "`!DEC$ VECTOR ALIGNED`" and "`!DEC$ VECTOR NONTEMPORAL`" [Int03].

Directive !DEC$ IVDEP

The `IVDEP` directive instructs the compiler to treat assumed vector dependencies as independent. In *LBMKernel*'s RISC version the `pdf` vector in write back loops could depend itself, if `tNow` has an appropriate negative value so that loop unrolling and local instruction scheduling cannot be performed safely.

Directive !DEC$ VECTOR ALIGNED

The `VECTOR ALIGNED` directive overwrites the efficiency heuristics of the vectorizer. It instructs the compiler to use aligned data movement instructions for all array references. Therefore Intel's User's Guide for Fortran compilers [Int03] recommends to use it with care, because instructing the compiler can cause a run-time exception when some of the access patterns are unaligned. That is the reason why RISC collision routine with LIJK data layout produces a segmentation fault if it is compiled with this directive (Since in LIJK data layout the 19 directions are first (see section 4.1), the dimension i is unaligned, because it lies not on addresses which are a multiple of 16).

Directive !DEC$ VECTOR NONTEMPORAL

`VECTOR NONTEMPORAL` directives advice the compiler to perform stores on Pentium 4 as streaming stores (non-allocating). By using the *movntps* SSE instruction, the written data is not stored in cache but "by-passed" directly to memory. Unfortunately, the advantage is noticeable just for large loops (as [Int03] points to).

3.4.3 Other Compiler's Options

The flags and directives mentioned above are mainly applicable for Intel's Compiler in 32-Bit version. But there are similar options for other compilers, too. The IA64 Intel compiler performs vector alignment and architecture specific optimization automatically. In addition to `IVDEP` directive there is a flag named `-ivdep_parallel`, which has to be used at the same time. Also the PGI compiler supports nontemporal stores (with flag `-Mnontemporal`). To specify the target architecture `-tp k8-64` was used (see Appendix F for used flags for different compilers).

Chapter 4

Memory Related Optimizations

The main approach to reduce the effect of the performance gap between processor and memory on modern micro architectures is the use of several hierarchically staged memories of different sizes and performance. Since high bandwidth and short latency are expensive, caches are smaller the faster and the closer to the CPU they are. Figure 4.1 shows a schematic overview of a typical hierarchy of memory levels. If the CPU loads data into its registers, it passes the request to the 1st Level Cache, also called L1 Cache. If the data is not available, a so called cache miss occurs. To minimize the influence of memory latency, the caches load a whole cache line from memory when a cache miss occurred.

Cache misses can be classified by the reason of occurrence. While there are compulsory misses, which are inevitable, one also differentiates between capacity misses, conflict misses and write misses. In direct mapped caches conflict misses occur, when data is loaded from memory, which is mapped to a location in cache where other valid data already resides that has to be replaced. In set associative caches there are several locations for storing the elements. Here, a conflict miss occurs, if all sets already contain valid data (see Figure 4.2). Write misses occur, when data is written to cache, which does not reside there. For handling such requests, there are different strategies. Caches that are write-back organized have to perform write-allocate, which results in an additional load from memory. Write-through organized caches exist in two types: Some have to perform write-allocate, others by-pass the data without storing it on their own.

Accordingly, in case of write-back caches, a read miss can cause a store and a load of a whole cache line from memory, if the miss was a conflict or a capacity miss. But stores result always in a load and a store to memory, if the data does not exist in cache.

For the development of software, thus it is important to take the cache organization of the architecture into account. The aim is to use the bandwidths of caches and to avoid loads and stores from memory. Ideally, the number of memory transfers will be reduced to the minimum, while all other data operations happen on local cache data. Since most of today's architectures have write-back organized caches, the code should be optimized such, that write misses occur as rare as possible and conflict and capacity misses are preferably avoided. Hence the aim of optimization is to have the actually required data as close to the CPU as possible and to keep the data in the cache as long as it is needed by CPU. Thus, two types of locality can be distinguished: temporal and spatial locality. Temporal locality means that data which is currently used will be used again in the near future more likely than later. Spatial locality means that data which is currently used and data in close neighborhood will more likely be used in the near future than data that is farer away. Achieving temporal locality globally in a program is often more difficult than to ensure spatial locality.

In conclusion, the best strategy would be *stride-1-access* (also called *unit stride access*) for both

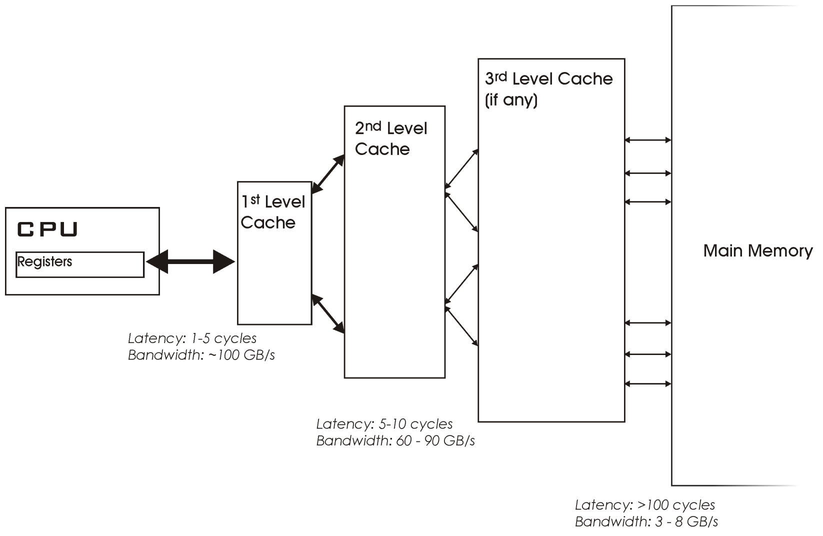
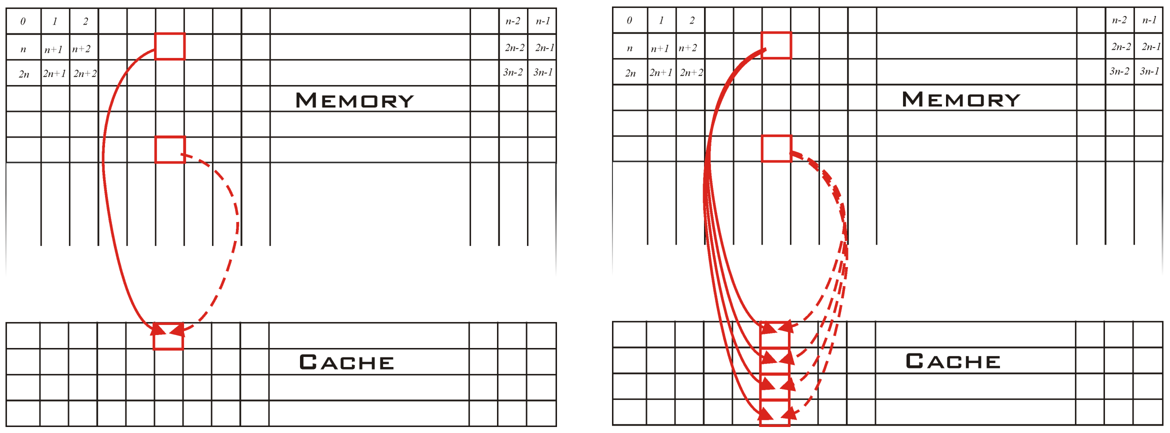


Figure 4.1: Schematic overview of different hierarchy levels of memory. The higher the level of memory is, the longer its latency and lower its bandwidth is. The closer to CPU the faster but smaller the caches are. The denoted magnitudes of bandwidth and latency are common for today's scalar architectures.



(a) Schematic description of a direct-mapped cache. Each memory location has a specific target location in cache. Since the cache is smaller than memory, there are many memory locations that are mapped to the same cache location.

(b) Schematic description of a 4-way set associative cache. Each memory location can be mapped to four cache location, but many memory locations map to the same four cache locations.

Figure 4.2: Different cache mapping strategies (adapted from [GHo01]).

reading and writing data, which results in full use of data so that no data is loaded in vain. When code and data can be rearranged to achieve stride-1-access, the obtained spatial locality minimizes the number of cache misses, because each loaded and stored element of a cache line was used. If data is used only once, or only at the time it resides in cache (temporal locality), optimal use of data is achieved and it was taken advantage of the cache hierarchy, optimally.

The following sections present techniques that increase both spatial and temporal locality for *LBMKernel*. It is explained in detail why these methods have an effect on code performance. The figures in this chapter base on sources, which were provided by courtesy of Klaus Iglberger.

4.1 Data Layout Optimization

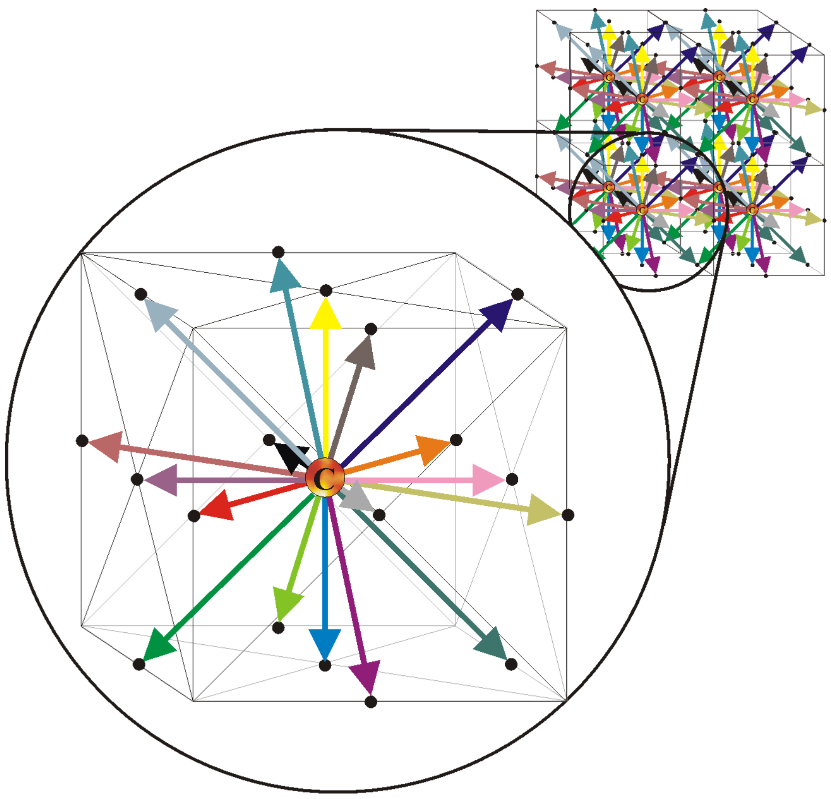
For optimization of *LBMKernel*, an important decision is the data layout of the `pdf` array. Concerning the spatial locality of the memory accesses of *LBMKernel*, three different layouts have been implemented. The `pdf` array consists of four or five dimensions, respectively, where one dimension stores the 19 values of the particle velocity distribution and three dimensions denote the coordinates of cells. Since the origin values have to be preserved until a full Lattice Boltzmann step is completed, simply two grids can be used, where the updated values after propagation are stored in the second grid while the origin values reside in the first one (see section 4.2). For this approach a fifth dimension is used, which selects between the two grids.

4.1.1 LIJK Data Layout

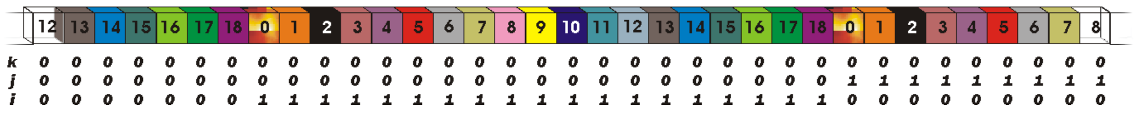
For the D3Q19 model the most intuitive data layout (which also was used in [Igl03]) is to use the first dimension of `pdf` array to store the 19 values of particle velocity distribution for each cell. The cell itself is identified by the three following indices in array. Here, this layout is called LIJK data layout, where l denotes the 19 directions and i, j and k stand for the three dimensions. Figure 4.3 shows the conception of the LIJK data layout: The 19 directions that belong to one cell are contiguous in memory. Such sets of 19 `double` values are subsequently ordered according to the three dimensions i, j and k, which stand for x, y and z, respectively.

In *LBMKernel* the realization of LIJK data layout and the addressing is done by the `IJKL_R` and `IJKL_W` macros, which are both defined as “`#define IJKL_R(i,j,k,l,tN) l,i,j,k,tN`” in case of using the implementation on two grids (see Algorithm D.1).

The LIJK data layout provides spatial locality for particle velocities of one cell. This is useful for that part of collision routine, where the actual `pdf` values of the current cell are read. When the access of the first direction causes a compulsory cache miss, some of the next velocities that have to be used next will be loaded, too. Therefore, these directions reside in cache already, when they have to be read. The 19 consecutive words usually occupy more than one cache line. That results in loading of averaged two cache lines for one cell, where due to the relative displacement of the directions also three or only one cache line may be fetched. However, the main disadvantage of this memory layout appears in propagation: When writing new values back in array with LIJK layout, the adjacent cells lie far away in memory, since the target cells are located in the second grid. The effect is, that theoretically each single store in adjacent cell causes an expensive cache write miss (see scattered write accesses in Figure 5.18(b) on page 54). Simulation with *kcachegrind* indicates, that this applies not always. Depending on the system size there are cases where the pushing of some



(a) Conception of LIJK data layout. The 19 velocities per cell are stored for each lattice site subsequently in order of dimensions *i*, *j* and *k*.


12 13 14 15 16 17 18 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 0 1 2 3 4 5 6 7 8
k 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
j 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
i 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0

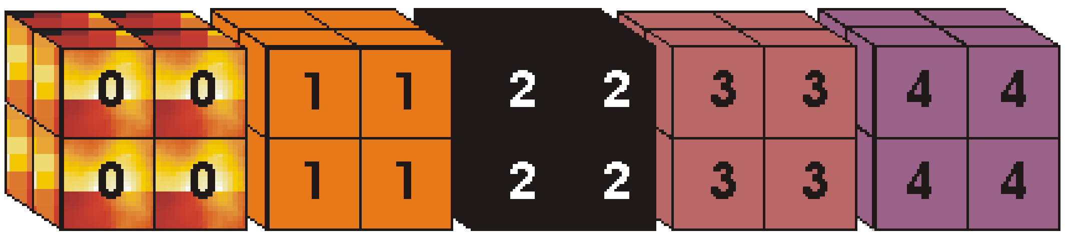
(b) Order of data in memory. The sets of 19 directions of a cell lie consecutively in memory ordered by the *ijk*-indexed position of lattice site in grid.

Figure 4.3: Illustration of LIJK Data Layout.

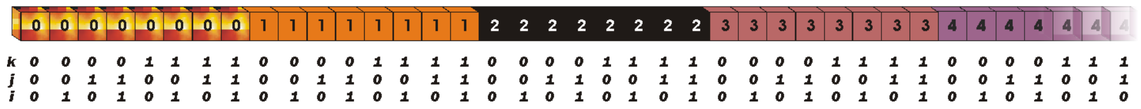
cell's updated values causes no cache miss: If the cache is large enough to hold the 19 velocities of the whole dimension *i* for the preceding, current and succeeding *j*-iteration, high cache reuse is possible. Thus, the performance will be dependent on the length of dimension *i*.

4.1.2 IJKL Data Layout

Comparing the IJKL data format with LIJK data format, the only difference is the position of the 19 directions in the order of the array indexes. This small change has a significant effect on memory layout. Now, the three dimensions *i*, *j* and *k* are close together, repeated 19 times, once for each direction. The result is, that all values for direction zero lie side by side for each cell in order of their coordinates (expressed by spatial indices) in memory, followed of all values for direction 1, and so on. Figuratively speaking, one could imagine 19 separated grids, where each grid contains the values of one direction for all lattice sites (see Figure 4.4).



(a) IJKL data layout can be imagined as 19 different small fields, each containing one direction, ordered in a three dimensional manner.


0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4
k 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1
j 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1
i 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

(b) Order of data in memory. For each of the 19 directions, the dimensions *i*, *j* and *k* lie consecutively in memory. Two directions of same lattice site are $k \cdot j \cdot i$ doubles apart.

Figure 4.4: Illustration of IJKL Data Layout.

When using the Two Grid memory layout, the IJKL data format is implemented by defining the `IJKL_W` and `IJKL_R` macros as "`define IJKL_R(i,j,k,l,tN) i,j,k,l,tN`".

Originally, the IJKL data format was introduced for the RISC version of *LBMKernel*. As mentioned in section 3.3, the principle idea was, to avoid the main handicap of the LIJK data layout. In terms of cache reuse, the LIJK data layout is terrible if applied to Lattice Boltzmann method, because of the high number of cache misses when writing new data to adjacent cells (see scattered accesses in Figure 5.18(b) on page 54). As outlined before, the goal is to reach unit stride access. Thus, if a cache line is fetched due to a cache miss, all data of the cache line will be used at least once, such that no data is loaded in vain. For reaching stride-1-access on both read and written data, the RISC version was split into several small loops, which run over the dimension *i*. When using the IJKL data layout, this is the first index, for which data lies in memory consecutively (Fortran data layout). Especially the RISC version profits by IJKL data layout. In first inner loop the `pdf` is read by unit stride access. Thus, only each 16th iteration causes 19 compulsory cache misses. All other loads can be saturated from data available in cache. In theory, the maximal performance had to be achieved, presumed, the cache can hold all 19 directions of the dimension *i* (and the additional temporary arrays). Because then the 19 write back loops would read values from cache in unit stride access. Also writing to the target locations occurs with stride-1-access. But each of these operations now cause a cache write miss again. For that, on Pentium 4 platforms the directive `!DEC$ VECTOR NONTEMPORAL` is included before every write back loop (see section 3.4 for details on this directive). The intention is, that these stores are by-passed on cache and this way cannot cause a cache write miss. In Standard version the IJKL data layout has even greater advantages: While there are equally often compulsory read misses when loading `pdf` data like in RISC version, this data lies definitely still in cache, when `ne`-variables are computed (In RISC version parts of this data could be thrown out of cache, if system size or length of *i*-dimension is too large due to the many temporary arrays with size of dimension *i*). In propagation of standard version the `VECTOR NONTEMPORAL` directive cannot be used. Thus, here every write operation causes a write miss. But since the memory usage of standard version is much less than RISC version's, the probability is high that the fetched cache lines due to these write misses still exist in cache when the next iteration of *i*-loop is performed. This results in a theoretic number of cache write misses equally to the number of read misses: 19 every 16th loop iteration. Since Vector version is nearly the same like standard version, the considerations made above are valid for Vector version, too, especially for scalar architectures. But even vector processors reach better performance with IJKL data layout in Vector version, as [ZWH04] showed with a Cray X1.

4.1.3 ILJK Data Layout

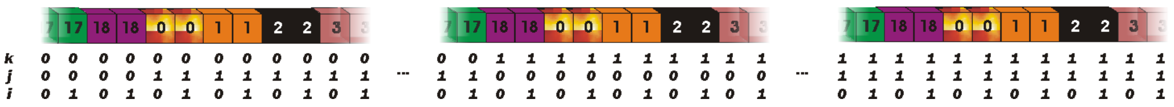


Figure 4.5: Order of data in memory when ILJK data layout is used. For each of the 19 directions the dimension i lies consecutively in memory. The sets consisting of 19 i-dimensions, are ordered by jk-indexed rows. Two directions of same lattice site are i doubles apart.

The big space between two directions in IJKL format increases the possibility that they lie on memory locations which conflict each other in cache. The distance of memory locations which are mapped to same set of cache line is the cache size divided by the number of its sets. In case of Pentium 4, the L2 cache is 8-way set associative with a set size of 64 KB. That means, there cannot reside more than eight memory locations, that lie a multiple of 64 KB apart each other, in cache at the same time. To avoid conflict misses, ideally the maximal size of the range in which memory accesses occur, had to be the cache size. Then, maximal spatial locality is attained. But when using the IJKL data layout, the memory locations that are read at the same time and near future, are widely remote, because all values of other j- and k-loops are in between, which would not yet be needed. The possibility is high, that the cache is fragmented: thus, several cache lines cannot be used during one iteration of j, because they are mapped to memory locations where data lies, that is needed only later (for other iterations of j).

To improve spatial locality, the ILJK data layout was developed. The directions are grouped by rows, that contain one whole dimension i and are indexed by a jk-combination. The effect is, that data that is needed for one i-iteration fits in cache without having conflicts more likely than in case of IJKL data layout due to the smaller range in memory accessed at the same time and higher spatial locality, respectively. Since the distances get shorter, the number of pages accessed at same time and near future, is less. Thus, there are less translations of logical to physical address to buffer, which results in less page faults (Translation Lookaside Buffer misses).

Figure 4.5 shows the order of data in memory. It is not possible to give a graphical representation, because the three dimension are not contiguous any more. That is also the reason, why vector version cannot operate with this data layout (The index m needs the dimensions consecutive in array). ILJK data layout includes all advantages of IJKL data layout. It also has unit stride access on i and therefore preloads the next 15 cells while having compulsory cache misses every 16th cycle in standard version.

4.2 Data Size Optimization

Besides adjusting layout of data also decreasing the size of data – without loss of information, of course – can improve performance, because it is possible to enhance spatial locality, which results in fewer cache misses. For this thesis two different implementations of grid data were realized. Besides the naive version with two grids, a compressed grid was designed, that needs only nearly half as much memory [Igl03]. Due to the data dependencies of the Lattice Boltzmann Method, it is not possible to reduce memory usage to the size of one grid, completely.

4.2.1 Two Grid Layout

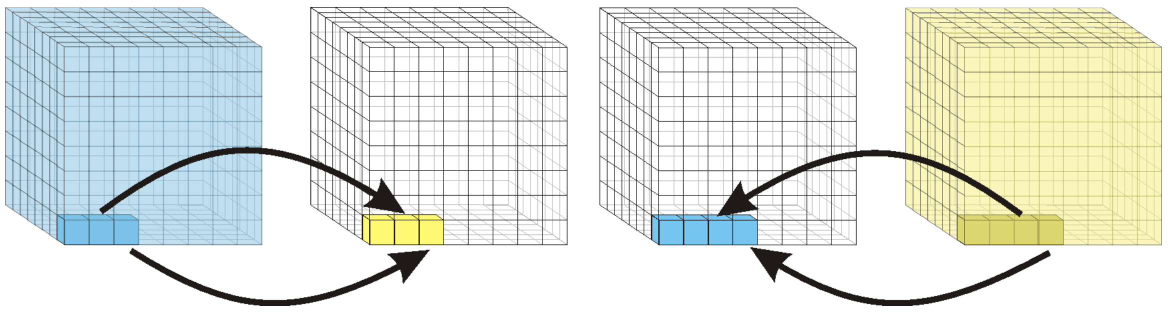

(a) Cells are updated successively from Grid 0 into Grid 1. (b) Cells are updated successively from Grid 1 into Grid 0.

Figure 4.6: Cell updates in Two Grid layout. After completion of a whole grid update, same procedure is done in reversed direction.

For Lattice Boltzmann method, each update of a lattice site needs the origin values of all adjacent cells. Since they have to be preserved until all depending cells are updated, the easiest and most intuitive way is the use of two grids. After a complete update of the whole grid origin and target grid are changed and the next time step of LBM is performed same way into opposite direction (illustrated by Figure 4.6).

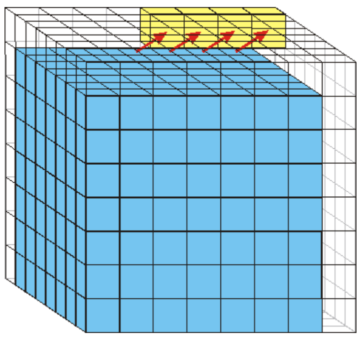
In *LBMKernel* the two grids are realized by a fifth dimension in `pdf` array, where `0` indexes the first, and `1` the second grid. The subroutines, that perform collision, receive two parameters from main routine: `tNow` and `tNext` (see Algorithm 4.1 for main routine and Algorithm D.1 for exemplary collision subroutine). `tNow` contains the value for addressing the grid with source values, `tNext` is the index of grid, which receives the new values. After returning from the subroutine, `main` switches the two grids and calls the subroutine again for the next time step.

Algorithm 4.1: Schematic concept of main routine, calling collision subroutine and switching grids

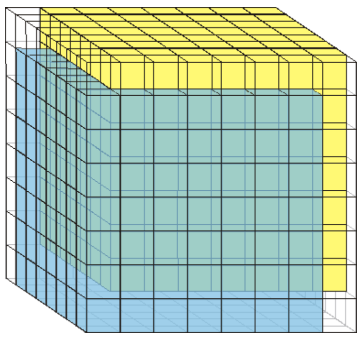
```
tNow=0
tNext=1

do timestep = 1, tmax

   call col_standard_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
   call bounceback_index_2G(pdf,tNext,barrierIdx,barrierCnt,iEnd,jEnd,kEnd,obstacleField)

   ! switch Source and Target Grid
   tNow=1-tNow
   tNext=1-tNext

enddo
```

The Two Grid layout has two important disadvantages: First, the read data and written data is far apart from each other in memory. This can cause many cache conflicts and page faults. Second, the extensive usage of memory: For storing a system with 100^3 cells, two grids of size 102^3 are needed (see Figure 1.2). Therefore the common system size of 100^3 needs 308 MB memory space only for `pdf`.

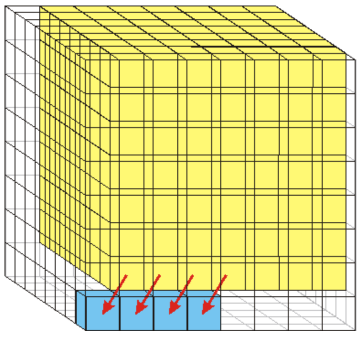
4.2.2 Compressed Grid Layout



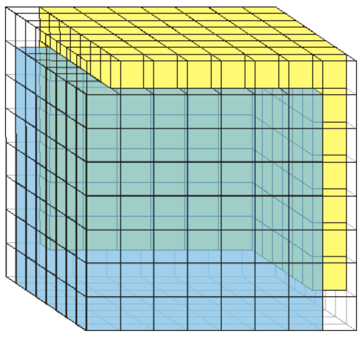
(a) Lower grid is updated and shifted cell by cell with $(+1,+1,+1)$ vector into upper grid.



(b) After complete time step the whole grid is shifted.



(c) In next time step, upper grid is updated and shifted cell by cell with $(-1,-1,-1)$ vector into lower grid.



(d) After second time step the whole grid is shifted back.

Figure 4.7: Update progress in Compressed Grid layout. The current grid is shifted by a displacement vector into second grid position. The direction is depending on current time step.

As mentioned before, it is not possible to perform Lattice Boltzmann method on only one grid, due to data dependencies. However, since the dependencies concern only adjacent cells, not all origin values have to be kept during the complete time step. Thus, it is possible to reduce the two grids to one of nearly half size. The resulting grid layout is called Compressed Grid [Igl03]. Figure 4.7 visualizes the idea of Compressed Grid: The grid is extended by one cell in each dimension, so that the whole system fits into it, and at top, right and back empty layers are added (Note that in this description ghost layers are included in system size, implicitly). The update procedure begins at the cell which is on top-right-backmost position. The updated values are written to same position, but shifted by displacement vector $(+1,+1,+1)$. Thus the target system is shifted in Compressed Grid technique by one cell into top, right and back direction. This update procedure is done in subsequently way for all cells of the lower grid, where new values are written to cells, which contained old values, but are not required any more. After a complete time step, all new values reside in the upper grid, which is in upper right back corner of whole Compressed Grid system. Figure 4.7(b) indicates, that in lower, left and front layer of Compressed Grid system still values of the old lower grid exist, which were not overwritten. In next time step (see Figure 4.7(c)), the same procedure is done in other direction: First the cell in lower left front corner of upper grid is updated and shifted by displacement vector $(-1,-1,-1)$ downwards in lowest left front cell of lower grid. After this second time step, all new values are stored in the lower grid, again, and the upper, right and back layer contain still old values of the upper grid, since they were not overwritten (Figure 4.7(d)). Since the D3Q19 model of LBM contains no particle velocity distribution values in direction $(-1,-1,-1)$ and $(+1,+1,+1)$, grid shifting in these directions is possible without conflicts with old values. For other models (like D3Q15 or D3Q27), only displacement vector had to be adapted (shifting by $(-2,-1,-1)$, for example). Thus, the implementation of Compressed Grid is relatively easy and independent of the model applied.

For implementation of Compressed Grid in *LBMKernel* the loops over k, j and i have to be traversed backwards from `End` to `1`, or forwards from `2` to `End+1` according to the current update direction, which depends on the time step. When writing the updated values to target cell, the displacement vector $(+1,+1,+1)$ or $(-1,-1,-1)$ has to be added, respectively. This can easily be realized by interpreting the values of parameters `tNow` and `tNext`. For integration purposes the functionality of Compressed Grid is included in same code as the Two Grid layout, by using preprocessor directives and macros. Algorithm 4.2 outlines the modifications of Standard version of collision subroutine.

For writing updated values into correct displacement direction, the `IJKL_W` macro is adapted. It adds the current `CompGridDir` to `i`, `j` and `k`, where `CompGridDir` is `+1` or `-1`. Additionaly it drops the fifth parameter, denoting the grid in Two Grid layout, since `pdf` array for Compressed Grid consists of only four dimensions (see different declarations of `pdf` in lines 40 and 42). The adaption of loop boundaries and loop direction is done before calculation of collide step: The loop boundaries are not constant any more, but variable. When the source for Two Grid layout is compiled (no -DCOMPRESSEDGRID flag specified by Makefile), the compiler recognizes that these variables only once set to their values `1` and `iEnd`, `jEnd` and `kEnd`, respectively, and will proceed its work as it does with constant boundaries. In case of compilation with COMPRESSEDGRID defined, the boundaries are set to `1..iEnd` and `iEnd+1..2` according to the values of `tNext` and `tNow`. In same way, the stepping of loops, denoted by `LoopDir`, is set to `+1` for counting upwards and `-1` for downwards, respectively. These calculations have to be performed once each time step. Since it is simple Integer arithmetic, it will not affect the performance. The body of loops is same as in Two Grid version. The different accessing of `pdf` array is managed by `IJKL_R` and `IJKL_W` macros, which are defined differently now. Since array `obstacleField` is bound to specific coordinates, the addressing has to be adapted depending on which grid is the actual one (see line 76).

Algorithm 4.2: Schematic concept of Compressed Grid implementation

```
 1 #ifdef IJKL_LAYOUT
 2 #ifdef COMPRESSEDGRID
 3 subroutine col_standard_ijkl_CG(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
 4 #define IJKL_R(i,j,k,l,tN) i,j,k,l
 5 #define IJKL_W(i,j,k,l,tN) (i+CompGridDir),(j+CompGridDir),(k+CompGridDir),l
 6 #else
 7 subroutine col_standard_ijkl_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
 8 #define IJKL_R(i,j,k,l,tN) i,j,k,l,tN
 9 #define IJKL_W(i,j,k,l,tN) i,j,k,l,tN
10 #endif
11 #else
12 #ifdef ILJK_LAYOUT
13 #ifdef COMPRESSEDGRID
14 subroutine col_standard_iljk_CG(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
```

```
#define IJKL_R(i,j,k,l,tN) i,l,j,k
#define IJKL_W(i,j,k,l,tN) (i+CompGridDir),l,(j+CompGridDir),(k+CompGridDir)
#else
subroutine col_standard_iljk_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
#define IJKL_R(i,j,k,l,tN) i,l,j,k,tN
#define IJKL_W(i,j,k,l,tN) i,l,j,k,tN
#endif
#else
#ifdef COMPRESSEDGRID
subroutine col_standard_lijk_CG(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
#define IJKL_R(i,j,k,l,tN) l,i,j,k
#define IJKL_W(i,j,k,l,tN) l,(i+CompGridDir),(j+CompGridDir),(k+CompGridDir)
#else
subroutine col_standard_lijk_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField,rho,acc)
#define IJKL_R(i,j,k,l,tN) l,i,j,k,tN
#define IJKL_W(i,j,k,l,tN) l,i,j,k,tN
#endif
#endif
#endif

  implicit none
  ...
  ! Different declarations of pdf for Two Grid layout and Compressed Grid layout
#ifdef COMPRESSEDGRID
  integer (I4B) :: CompGridDir
  real (dp), dimension(IJKL_R(0:iEnd+2,0:jEnd+2,0:kEnd+2,0:18,0)) :: pdf
#else
  real (dp), dimension(IJKL_R(0:iEnd+1,0:jEnd+1,0:kEnd+1,0:18,0:1)) :: pdf
#endif

#ifdef COMPRESSEDGRID
  !if tNow=0, tNext=1 -> shift downwards, calculate right upper grid, write into lower left
  !if tNow=1, tNext=0 -> shift upwards, calculate on grid in lower-left-corner

  CompGridDir=tNow-tNext  ! -1 if shift downwards, +1 if shift upwards

  !calculate Loop-Bounds according to direction:
  LBOUNDK = (tNow  * kEnd) + (tNext * 2)          ! =      2 or kEnd
  UBOUNDK = (tNext * kEnd) + tNext + tNow          ! = kEnd+1 or 1
  LBOUNDJ = (tNow  * jEnd) + (tNext * 2)          ! =      2 or jEnd
  UBOUNDJ = (tNext * jEnd) + tNext + tNow          ! = jEnd+1 or 1
  LBOUNDI = (tNow  * iEnd) + (tNext * 2)          ! =      2 or iEnd
  UBOUNDI = (tNext * iEnd) + tNext + tNow          ! = iEnd+1 or 1

  LoopDir = tNext-tNow  ! +1 if shift downwards, -1 if shift upwards
#else

  LBOUNDK = 1
  UBOUNDK = kEnd
  LBOUNDJ = 1
  UBOUNDJ = jEnd
  LBOUNDI = 1
  UBOUNDI = iEnd

  LoopDir = 1
#endif

  do k=LBOUNDK,UBOUNDK,LoopDir
     do j=LBOUNDJ, UBOUNDJ,LoopDir
        do i=LBOUNDI,UBOUNDI,LoopDir
#ifdef COMPRESSEDGRID
           if (.not. obstacleField(i-tNext,j-tNext,k-tNext)) then
#else
           if (.not. obstacleField(i,j,k)) then
#endif
             ...
           endif
        enddo
     enddo
  enddo
end subroutine
```

Regarding computational intensity, the additional additions in address computation of `pdf` caused by the macro, are not severe. First of all, the absence of fifth dimension saves an addition per `pdf` access. Furthermore, adding `CompGridDir` to `i`, `j` and `k` is done only once: the compiler computes base address and `i+CompGridDir`, `j+CompGridDir` and `k+CompGridDir` once, and then adds `LoopDir` on the dimensions each iteration. Thus, it is same calculation as in case of having no `CompGridDir` added (as used in Two Grid layout). Due to the need to add the displacement vector's components to the single dimensions, the m-loop of Vector version, which conclude all three dimensions in one index, makes the implementation of Compressed Grid for Vector version impossible.

The Compressed Grid is able to improve performance of *LBMKernel*. Since most of values are written to cells which were read short time before, there is a realistic chance that the corresponding cache lines are still in cache. Thus, writing the new values will cause less cache write misses. Besides, the memory locations from which is read and these to which is written, are closer together, which improves spatial locality, and, thus, reduces the probability of cache mapping conflicts. By using the Compressed Grid, the memory usage is much less than in case of using Two Grid layout. For the common system size of 100^3 it results in a memory usage for `pdf` of 158 MB, only.

4.3 Data Access Optimization

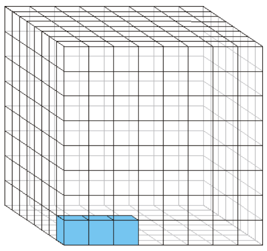
Since it is often not possible to have small strides for all referenced data in an algorithm, there are several common strategies to improve spatial locality. One of these techniques is loop blocking, which is used also in this thesis. The idea of blocking is to divide the loop into several smaller parts and compute then part by part (the so-called blocks). The size of blocks is commonly such that one block fits in cache and the algorithm can work with cache performance. However, the additional computational effort for the blocking loop, and the inevitable data transfer from and to memory, result often in getting minor performance increase. Of course, use of blocking is possible only if data is independent such a way that the problem can be partitioned without computational errors.

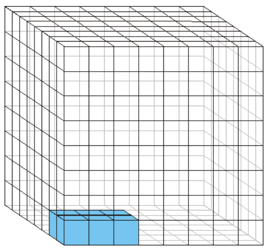
4.3.1 1D-Blocking

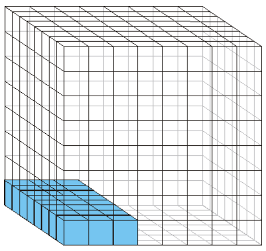
The technique of 1D-Blocking, also called line blocking, is commonly used when irregular domains are used and the innermost loop is long. In *LBMKernel*, when using LIJK format, the 19 directions lie irregularly in memory, so that line blocking can be useful (see [GHo01] for more details). But the main reason for introducing 1D-Blocking were the long and cache space wasting arrays of the RISC version. Blocking in one direction is done by chopping the innermost loop in several parts of a specific size (the block size), and performing the work for each of these parts (by introducing an additional loop around algorithm). However, this is reasonable only in cases where local data does not fit in cache.

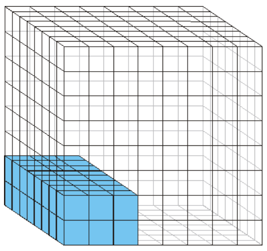
Figure 4.8 shows the process of 1D-Blocking: First, it starts as usual with updating cells subsequently in i-direction, but only as long as the block size is. Then the next i-row of length of block size is updated and so on, until j-loop is traversed once and one layer is updated. After having traversed whole k-loop, the next block is computed (Figure 4.8(e)), and so on, until the whole grid is updated.

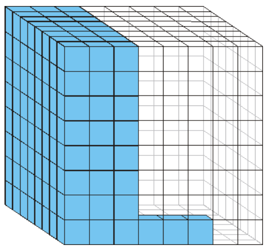
This process shows clearly the advantage for the RISC version: The 25 additional arrays are as long as the innermost loop (length of i-loop, the information that has to be stored for one iteration of j), which is shortened by blocking to the length of block size. Thus, the RISC version saves memory,

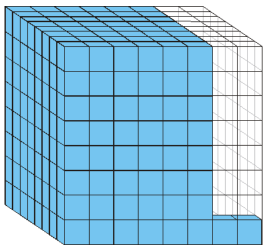

(a) First block of i-row is updated.


(b) Second block of i-row is updated (second j-iteration).


(c) After several iterations of j, first blocked layer is updated.


(d) Second layer is updated (second iteration of k).


(e) After several iterations of k, first block is updated. Same process for second block begins.


(f) Second block is updated. Last block may be smaller, and fills up grid.

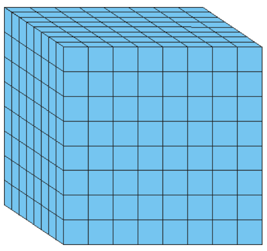
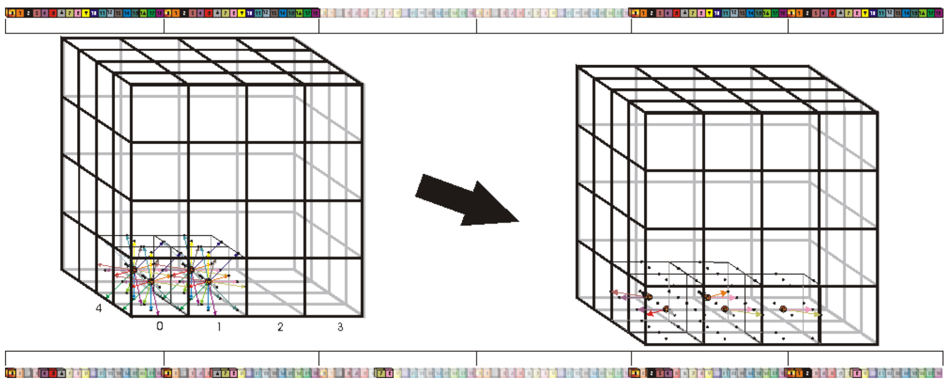

(g) Entire grid is updated.

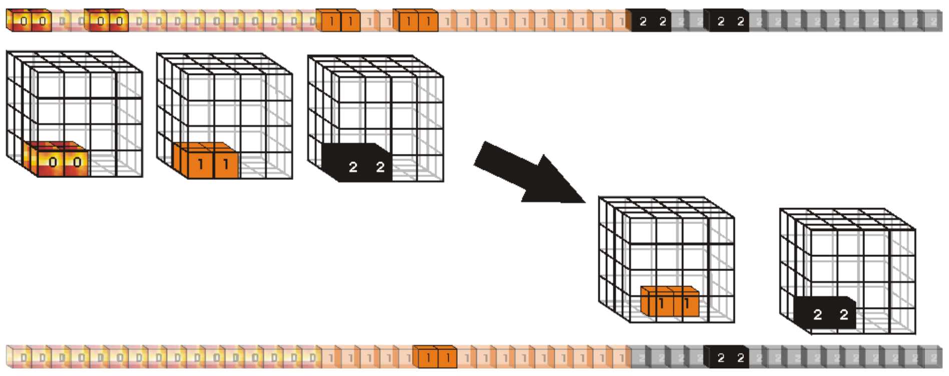
Figure 4.8: Update progress when 1D-Blocking is applied.

whereby the data for more cells fits in cache. When increasing the length of dimension *i*, the 1D-blocked RISC version should keep the good performance of smaller sizes for larger lengths than the nonblocked version, regardless of the used data and grid layout.

The implementation of the 1D-Blocking in *LBMKernel* is again done by macros, so that only slight modifications were required (see Algorithm 4.3). If the Makefile defines WITHBLOCKING, an additional loop in front of the usual loops is inserted (line 11), which counts the blocks. It receives the same boundaries as the innermost loop in non-blocking case, but a stride of BLOCKSIZE (which is defined by Makefile). The innermost loop (line 18) traverses the blocks, beginning with current index of blocking loop, and ending at the end of block, or, if the rest of the system is smaller than the block size, at the end of system (this is done by the “min”-statement of line 8). In case of using Compressed Grid, the boundaries have to be adapted to the direction of traversing (line 5 shows the choice of “min” and “max”, correspondingly).

1D-Blocking was implemented in *LBMKernel* for RISC and Standard version. For the Vector version, blocking is not reasonable, because the Vector version and vector processors benefit from long loops. Blocking would just shorten the loop and destroy the nature of vector optimization. Considering the different data layouts, 1D-Blocking can be advantageous and disadvantageous. Figure 4.9(a) tries to visualize what blocking means for LIJK data layout. For clearness, a block size of two in a 4^3 system was chosen. Of course, blocking is usually performed for large system sizes. For reading values of current cells, all values lie consecutively in memory. The gaps between accesses of first and second i-row are not advantageous, but are only side effects, regarding usual block sizes. When


(a) LIJK data layout: All values needed for one block lie consecutively in memory. When writing new values, many of them reside in cache already.


(b) IJKL data layout: The successive order is broken by blocking for read and write access.

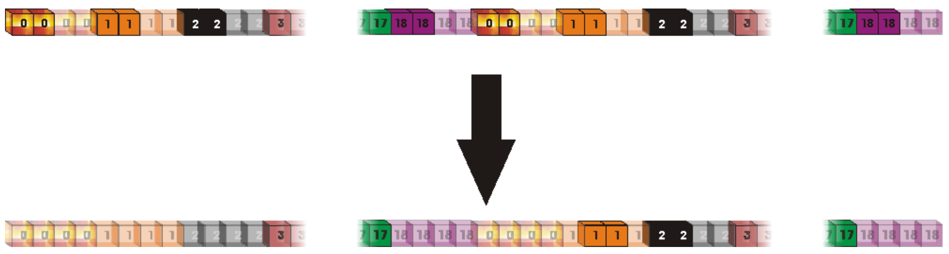

(c) ILJK data layout: Values for a cell lie close together, but not all values in between are used in an iteration of *j* and *k*.

Figure 4.9: Illustration of memory access with different data layouts, performing 1D-Blocking. For better understandability block size is chosen as two, here.

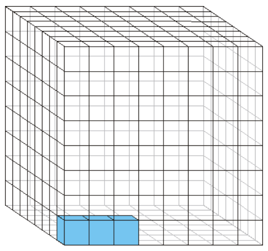
Algorithm 4.3: Additional statements for 1D-Blocking

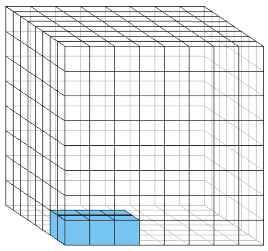
```
#ifdef WITHBLOCKING

#ifdef COMPRESSEDGRID
#define LBOUNDI2 ii
#define UBOUNDI2 min(UBOUNDI, ii+BLOCKSIZE−1) * tNext + max(UBOUNDI, ii−BLOCKSIZE+1) * tNow
#else
#define LBOUNDI2 ii
#define UBOUNDI2 min(iEnd, ii+BLOCKSIZE−1)
#endif
    ! Perform Block after Block, by stepping by BLOCKSIZE
    do ii=LBOUNDI,UBOUNDI, LoopDir*BLOCKSIZE

    do k=LBOUNDK,UBOUNDK, LoopDir
       do j=LBOUNDJ, UBOUNDJ, LoopDir

          ! count i from beginning of block
          ! to either end of block or end of system (if reached earlier)
          do i=LBOUNDI2,UBOUNDI2, LoopDir
#else
  do k=LBOUNDK,UBOUNDK, LoopDir
     do j=LBOUNDJ, UBOUNDJ, LoopDir
        do i=LBOUNDI,UBOUNDI, LoopDir
#endif

          ...

        enddo
     enddo
  enddo
```

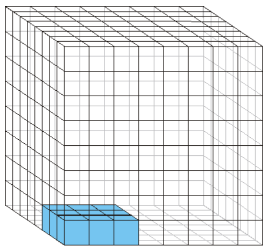
writing the values back (for example in second grid, when using Two Grid layout), spatial locality is increased: The stores of first i-row fetched cache lines, that could be still in cache, if second i-row writes new values in these cells, because dimension of i is shorter than in nonblocked case. In contrast, when using the IJKL data layout (see Figure 4.9(b)), there is no gain in spatial locality. The advantage of unit stride access over whole dimension i that IJKL data layout provides is lost. Because of 1D-Blocking, there are wide gaps between accessed memory locations. Same issue appears when writing new values into second grid. While Figure 4.9(a)) shows, that writing from cell number 0 to cell number 4 (direction 2 = North) writes probably to same cache line as writing from cell number 1 into cell number 4 (direction 3 = North-West) or from cell number 8 to cell number 4 (direction 6 = South), Figure 4.9(b)) makes obvious, that the target memory locations of cell number 4 are in different cache lines. The same is true for ILJK data layout. Since these layouts are optimized for unit stride access in long i-loops, blocking of i-loop leads to decline in spatial locality and, thus, in performance (see results in chapter 5).

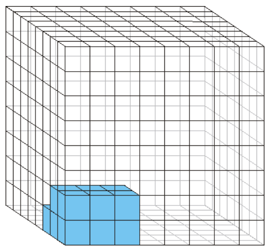
4.3.2 3D-Blocking

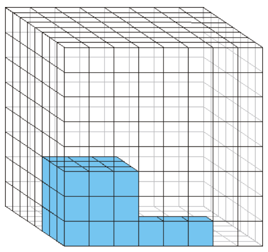
The idea of 3D-Blocking is – similar to 1D-Blocking – the increase of spatial locality, when large data sizes are used. The reason why 3D-Blocking could be better than 1D-Blocking is temporal locality. When the other two directions are blocked, too, the temporary system size is small enough, that all data, that is needed for this block, fits in cache. This data has not to be accessed another time until the whole grid is updated (except for side effects on boundaries of blocks). While the same is true for 1D-Blocking, 3D-Blocking owns the advantage, that data, that is fetched in cache due to writing updated values, is more probably still in cache, when this data is needed in the write back process of other adjacent cells, because due to the blocking in three dimensions, these cells will be accessed

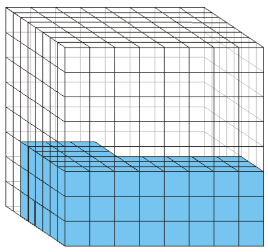

(a) First, update process in *i*-direction is performed for the length of block size for *i*-dimension.

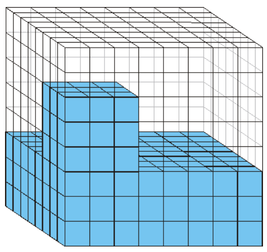

(b) With each iteration of *j*-loop, one *i*-row is updated.


(c) After *j* reached block size for *j*-dimension, one blocked layer is updated.


(d) The second iteration of *k* updates the second layer of block.


(e) After *k* reached block size for *k*-dimension, one block is updated and same process begins for next block.


(f) The three blocking loops update block after block in same order.


(g) The grid is updated block wise...

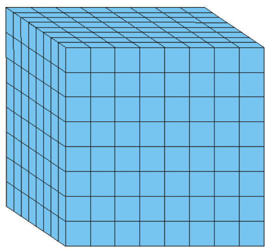

(h) ...until entire grid is updated.

Figure 4.10: Example for update progress when 3D-Blocking is applied. Here, the block sizes for the three dimension are of same value 3.

earlier. Thus, the temporal locality is in case of 3D-Blocking “more temporal local” than in case of 1D-Blocking. For clearness, Figure 4.10 shows the update process when 3D-Blocking is used. Compared to 1D-Blocking, the blocks are also limited in *j*- and *k*-dimension. But overall progress is the same: By updating block by block, the whole grid is updated subsequently. Here it is clearly observable, that due to more compact blocks (not in size but in extension ratios) temporal and spatial locality are increased.

Of course, this consideration is again only true for the LIJK data layout. When using the IJKL or ILJK data layout, which are optimized for long unit stride access on dimension *i*, 3D-Blocking will worsen the data locality as well as 1D-Blocking, and, thus, will degrade performance.

Implementation of 3D-Blocking in *LBMKernel* is straightforward (see Algorithm 4.4). Following the principle in the 1D-Blocking case, now for all three loops blocking is performed. For being able to use either 1D-Blocking or 3D-Blocking, the Makefile can switch via defining BLOCK3D. The three blocking loops are ordered, as already mentioned in 3D-Blocking process description (see Figure 4.10), same as dimension loops: first *k*, then *j* and last *i*. Lines 1 to 10 define block sizes for *j*- and *k*-direction, if the Makefile does not specify them.

Algorithm 4.4: Additional statements for 3D-Blocking

```
#ifdef WITHBLOCKING
#ifdef BLOCK3D
#ifndef BLOCKSIZEK
#define BLOCKSIZEK BLOCKSIZE
#endif
#ifndef BLOCKSIZEJ
#define BLOCKSIZEJ BLOCKSIZE
#endif
#endif
#endif

#ifdef WITHBLOCKING
#ifdef COMPRESSEDGRID

#define LBOUNDK2 kk
#define UBOUNDK2 min(UBOUNDK, kk+BLOCKSIZEK-1)*tNext + max(UBOUNDK, kk-BLOCKSIZEK+1)*tNow
#define LBOUNDJ2 jj
#define UBOUNDJ2 min(UBOUNDJ, jj+BLOCKSIZEJ-1)*tNext + max(UBOUNDJ, jj-BLOCKSIZEJ+1)*tNow
#define LBOUNDI2 ii
#define UBOUNDI2 min(UBOUNDI, ii+BLOCKSIZE-1)*tNext + max(UBOUNDI, ii-BLOCKSIZE+1)*tNow

#else

#define LBOUNDK2 kk
#define UBOUNDK2 min(kEnd, kk+BLOCKSIZEK-1)
#define LBOUNDJ2 jj
#define UBOUNDJ2 min(jEnd, jj+BLOCKSIZEJ-1)
#define LBOUNDI2 ii
#define UBOUNDI2 min(iEnd, ii+BLOCKSIZE-1)
#endif

#ifdef BLOCK3D
      do kk=LBOUNDK,UBOUNDK, LoopDir*BLOCKSIZEK
      do jj=LBOUNDJ,UBOUNDJ, LoopDir*BLOCKSIZEJ
#endif
      do ii=LBOUNDI,UBOUNDI, LoopDir*BLOCKSIZE
#ifdef BLOCK3D
      do k=LBOUNDK2,UBOUNDK2, LoopDir
         do j=LBOUNDJ2,UBOUNDJ2, LoopDir
#else
      do k=LBOUNDK,UBOUNDK, LoopDir
         do j=LBOUNDJ, UBOUNDJ, LoopDir
#endif
            do i=LBOUNDI2,UBOUNDI2, LoopDir
#else
   do k=LBOUNDK,UBOUNDK, LoopDir
      do j=LBOUNDJ, UBOUNDJ, LoopDir
         do i=LBOUNDI,UBOUNDI, LoopDir
#endif

            ...

         enddo
      enddo
   enddo
```

Chapter 5

Results

In the previous two chapters the basics for different optimization strategies were discussed. This chapter will present the actual results. First, the most important architecture dependent effects of the different memory layouts and blocking techniques are shown. After that, a short presentation of results of profiling and cache simulation is given. Then exemplary outcomes of *MemPHis*, the memory visualizer that was developed for this thesis, are presented, followed by the details about performance differences due to the use of different compilers.

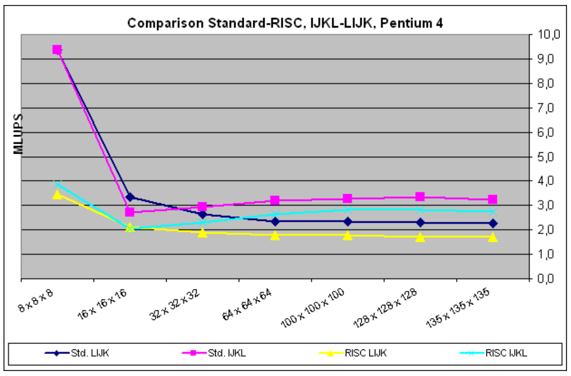
5.1 Results of Memory Related Optimizations

The following sections present the results of optimization efforts related to memory bandwidth. Starting with an overview of general results, which are valid for all test platforms, it is shown, which optimization efforts lead to reasonable performance improvements, while efforts for implementing other techniques can be saved. Sections 5.1.2 to 5.1.4 present individual results and exceptions concerning the different processor architectures, including comparisons between the Pentium 4 based platforms (Test Platform 1 and 2), the Itanium 2 platforms (Test Platform 3 and 4) and the different compilers for Opteron based Platform 5 (Intel IA32 and Portland compiler). For configuration of the test platforms refer to Appendix E. A complete collection of charts with results for each combination of optimization technique and test platform is listed in Appendix A.

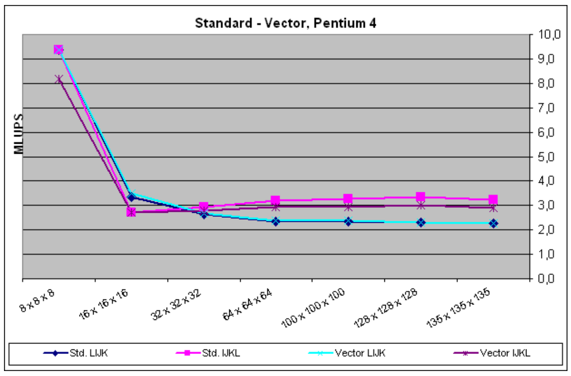
For easier investigation of impacts by 1D-Blocking (and due to limited memory space on some test machines), the charts relating to 1D-Blocking optimization technique use long systems. There, the system size is increased from 8^3 to 100^3 in cubic way. Above 100^3 the overall system size stays constant: While the dimension i is getting larger (up to about $10,000$), the other two dimensions are decreased accordingly. Such extremely long systems are uncommon. However, by applying the presented optimization techniques, it is possible to think about chopping a large system into several longer, non-cubic portions, when using parallelized codes, for example.

5.1.1 General Results

Generally, all architectures show high performance for small system sizes like 8^3, where all data fits in the caches (*cache effect*). Because of RISC routines' additional arrays the cache effect is significantly less than for others. For larger system sizes it is clearly recognizable that LIJK data layout is worst for all three versions (Standard, Vector and RISC), and the IJKL data layout is much better due to the unit stride on i (see Figure 5.1(a)).

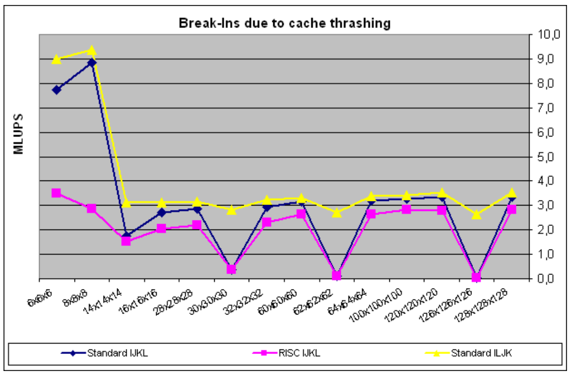


(a) Comparison of IJKL and LIJK data layout

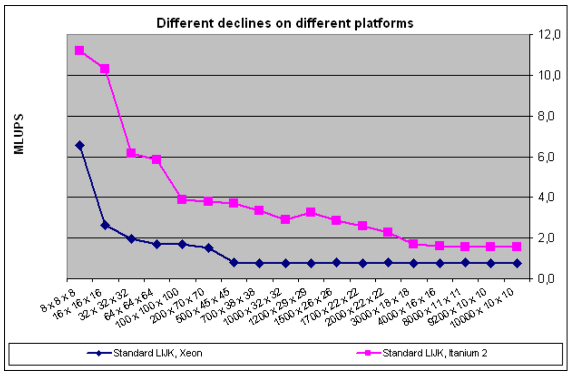


(b) Comparison of Standard and Vector version

Figure 5.1: Standard, Vector and RISC version on Test Platform 2 (Pentium 4, 3.0 GHz).

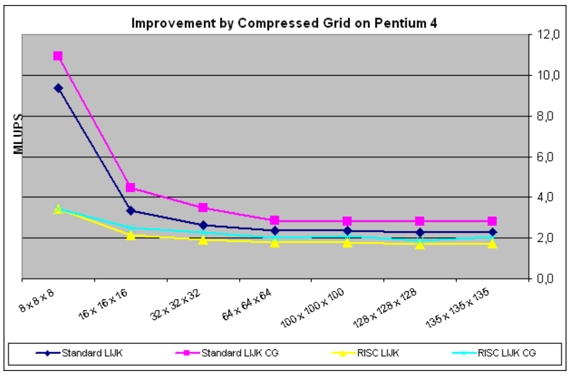


(a) Performance break-ins due to cache thrashing on Platform 2 (Pentium 4, 3.0 GHz)

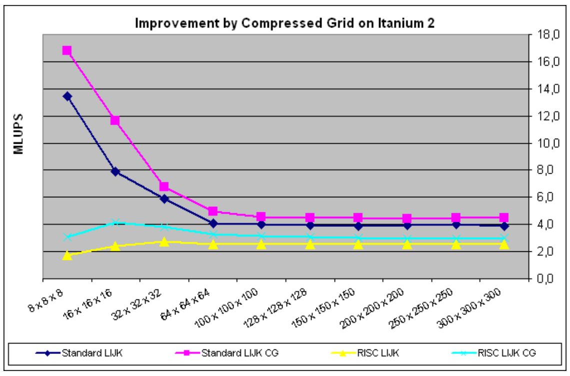


(b) Decline of LIJK data layout on Test Platform 1 (Xeon, 2.66 GHz) and Test Platform 3 (Itanium 2, 1.4 GHz)

Figure 5.2: IJKL and LIJK data layout influenced by cache issues.



(a) Test Platform 2 (Pentium 4, 3.0 GHz)



(b) Test Platform 3 (Itanium 2, 1.4 GHz)

Figure 5.3: Performance improvement by Compressed Grid technique.

In addition to the higher performance on Pentium 4 based platforms the ILJK data layout evades the cache conflict problem of IJKL data layout: When using domain sizes with dimensions of power of two the 19 directions are mapped to a small set of cache lines. Hence, cache conflict misses in an enormous number occur (*cache thrashing*), which results in poor performances (see Figure 5.2(a)). Note that the system sizes where cache thrashing occurs are smaller by two than the powers of two, because the domain contains additional ghost layers in each direction (see Figure 1.2).

As mentioned in section 4.1.1, the performance of the LIJK data layout depends on the length of dimension *i*. This can be seen in Figure 5.2(b) where the length of the system is continuously increased. Owing to different cache sizes, the performance drop occurs at different values of *i*. For short domain lengths the data of three full j-iterations and both grids fits into the cache. In theory, this is true for lengths of dimension i less or equal to

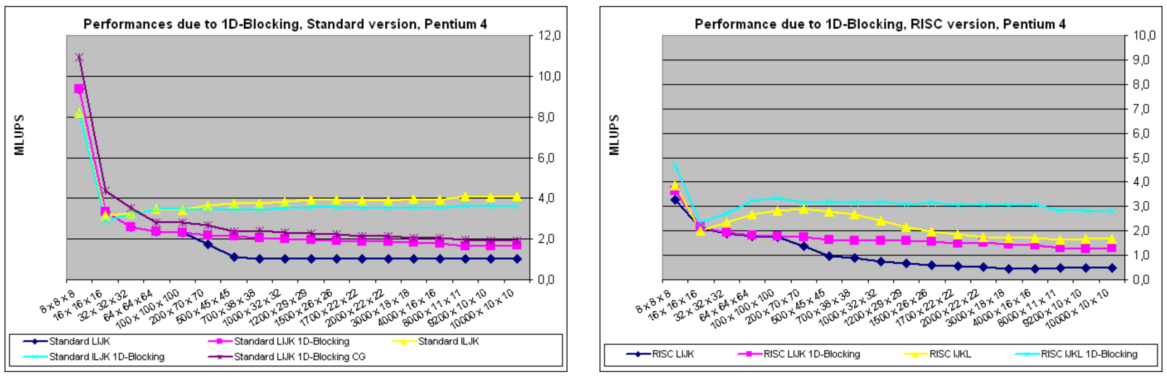
$$\text{maximum length of } i = \frac{\text{Cache size in Bytes}}{2 \cdot 3 \cdot 19 \cdot 8 \text{ Bytes}}. \tag{5.1}$$

Surely, the decrease will start earlier because of associativity and write-back behavior of the cache.

For all scalar architectures used it seems to be regardless whether the problem is expressed in one or three loops: The Vector versions have the same performance as the corresponding Standard versions (see Figure 5.1(b)). Since scalar architectures do not have wide vector pipes, they do not benefit from the increased loop length like vector processors. In addition, the overhead for starting loops obviously is negligible.

Compressed Grid Layout

The Compressed Grid technique decreases the system size and therefore the distances between read and written data. This increase of spatial locality leads to a significant performance increase for LIJK data layout only (see Figure 5.3 for different versions and architectures). The Opteron based Test Platform 5 experienced no enhancement by this technique (see section 5.1.4). The distributed shared memory (DSM) machine Test Platform 4 profits by Compressed Grid on system sizes beyond the local memory size (see section 5.1.3).



(a) Standard versions (b) RISC versions

Figure 5.4: Increase and decrease of performance due to 1D-Blocking on Test Platform 2 (Pentium 4, 3.0 GHz).

1D-Blocking

As predicted in section 4.3.1, the 1D-Blocking technique could not enhance the IJKL and ILJK data layout for the Standard version, because their advantage of stride-1-access does not depend on the length of the i-dimension. In contrast, some architectures show a degradation of performance. However, the LIJK data layout for the Standard and RISC version is improved: The decline caused by a long dimension i is removed and the curve's downtrend is replaced by a constant behavior in the graph (see Figure 5.4(a)).

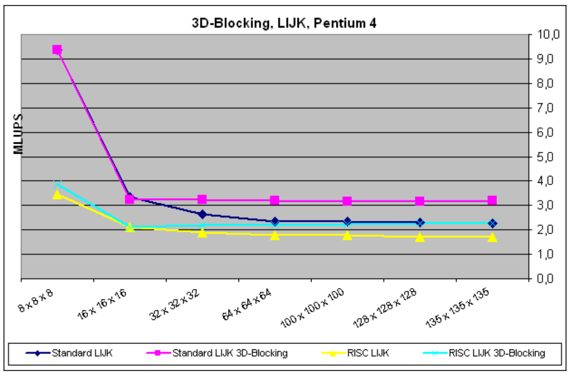
Originally, the aim of 1D-Blocking was to improve the RISC version by reducing the amount of wasted cache space due to its long temporary arrays. In fact, on systems with small caches all RISC versions experienced much improvement (even IJKL and ILJK), which is shown in Figure 5.4(b)). In contrast, on machines with large caches, only RISC LIJK version gets better (see Figure 5.11 in section 5.1.3).

To achieve enhancement by 1D-Blocking, the block size must have a size which reduces the dimension i to a length which allows high cache reuse by Equation 5.1. Since the smallest cache on the platforms used is on the Pentium 4 machine, the blocking factor was adapted to this cache size. With 512 KB in theory high cache reuse is possible up to a length of about 570. To be safe from side effects caused by associativity and the write-back strategy, the block size was chosen to 100.

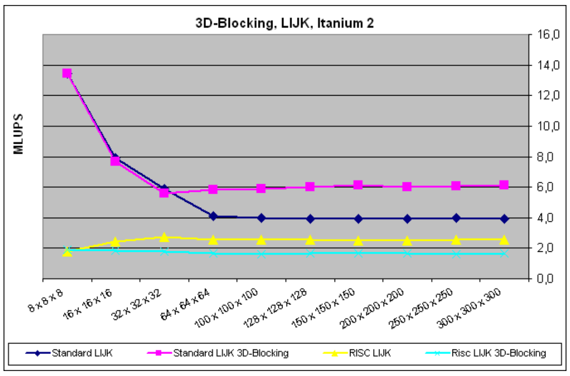
The increase of performance when combining 1D-Blocking technique with Compressed Grid layout is not mentionable. Except for the Standard LIJK version, which is slightly improved (see Figure 5.4(a)) on all platforms beside the Itanium 2 based Platforms 3 and 4 (section 5.1.3).

3D-Blocking

The purpose of the 3D-Blocking technique is to increase spatial and temporal locality, which provides best results, if the blocks are of compact size (in terms of extension ratios). It is assumed, that the optimal block geometry has a quadratic base. For this reason, a cubic block size was chosen. The overall size of the block must not exceed the cache size, so that the update of the current block can benefit from the cache effect. Since the performance on all platforms is best with a system size of 8^3,

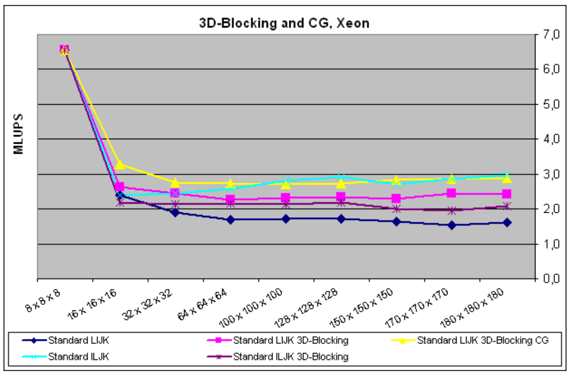


(a) Test Platform 2 (Pentium 4, 3.0 GHz)

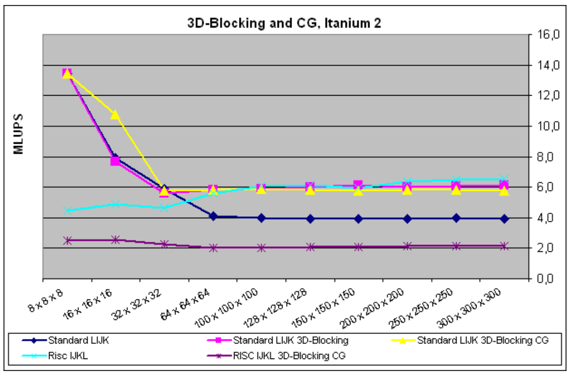


(b) Test Platform 3 (Itanium 2, 1.4 GHz)

Figure 5.5: Increase and decrease of performance due to 3D-Blocking for Standard and RISC version.



(a) Test Platform 2 (Pentium 4, 3.0 GHz)



(b) Test Platform 3 (Itanium 2, 1.4 GHz)

Figure 5.6: Increase and decrease of performance due to 3D-Blocking and Compressed Grid for Standard and RISC version.

this size was chosen for block size.

As section 4.3.2 denotes, 3D-Blocking is able to improve the LIJK data layout, only. In fact, performance of Standard LIJK version rose by 50% on all platforms (see Figure 5.5(a)). Also RISC LIJK improved significantly, except for the Itanium 2 based platforms (see Figure 5.5(b) and section 5.1.3). Due to the loss of unit stride access the IJKL and ILJK versions dropped vastly in performance (Figure 5.6(a)).

The Compressed Grid layout in combination with the 3D-Blocking technique leads to an additional improvement for Standard LIJK 3D-blocked version (except for Itanium 2). However, in most cases the Standard LIJK 3D-blocked version with Compressed Grid reaches a performance level, which is already obtained by the nonblocked versions with IJKL or ILJK data layout (see Figure 5.6). So, all optimization effort concerning blocking and Compressed Grid can be saved when using a data layout with unit stride on *i*. Only the Opteron based Test Platform 5 is an exception from this rule. Here, the 3D-Blocking technique in combination with Compressed Grid layout could enhance the Standard LIJK version to the overall best version on this platform (see section 5.1.4).

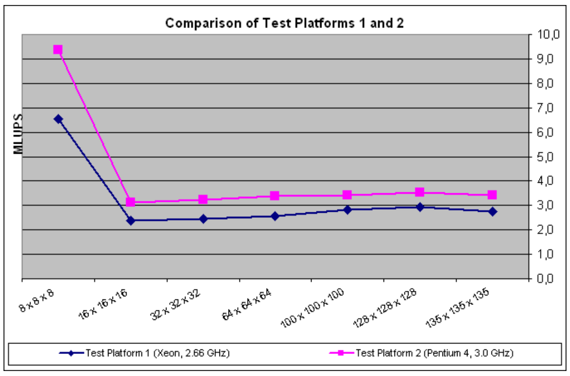


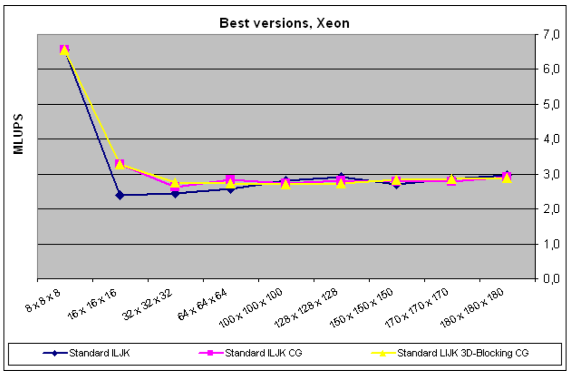
Figure 5.7: Comparison of Standard IJKL on Test Platform 1 (Xeon, 2.66 GHz) and Test Platform 2 (Pentium 4, 3.0 GHz).

5.1.2 Xeon / Pentium 4

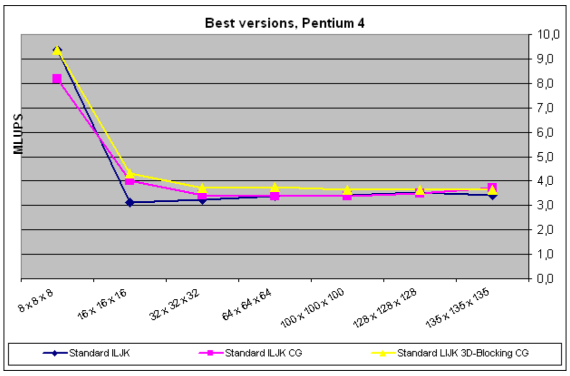
The two Pentium 4 based machines Test Platform 1 (Xeon, 2.66 GHz) and Test Platform 2 (Pentium 4, 3.0 GHz) are equipped with processors of different clock rates and memory bandwidth. While it is obvious, that Test Platform 2 delivers better results, it is to determine whether this improvement is due to the faster processor (3.0 GHz instead of 2.66 GHz) or the higher memory bandwidth (6.4 GB/s compared to 4.3 GB/s). Figure 5.7 gives a comparison between the two machines: Performance results of Test Platform 2 are about 30% higher (even for system sizes where performance is bound to memory bandwidth). Considering the ratios of clock rate (13%) and memory bandwidth (49%), surely the faster memory system is more responsible for the advance.

In Figure 5.3(a) can be seen that the Compressed Grid layout improves performance of the Standard LIJK version by about 30% on Pentium 4 based platforms. According to Equation 5.1 the decline of the versions with LIJK data layout is around $i = 500$ due to the cache size of 512 KB. The 1D-Blocking technique improves the lower performance by 100% to a constant curve progression (as denoted in Figure 5.4(b)). Even RISC IJKL and ILJK show an enhancement of about 60% in this area. Nevertheless, the RISC versions with stride-1-access on dimension i experience the drop in the nonblocked case only because of the long temporary arrays. The decline of RISC versions occurs when the 39 arrays with length of dimension i of third loop (see section 3.3.2) exceed the cache space (which is theoretically at $1,700$ on Pentium 4). In the end, the nonblocked Standard versions with ILJK layout (on Test Platform 2 with Compressed Grid layout) offer best performance for long systems on Pentium 4 based platforms.

The 3D-Blocking technique is able to increase the performance of Standard and RISC LIJK versions by about 40% to 50% (see Figure 5.5(a)). Additional use of Compressed Grid layout improves the results. Even the performance reduction of ILJK and IJKL data layouts is less. On Test Platform 2 (Pentium 4, 3.0 GHz) the enhanced Standard LIJK 3D-blocked version due to Compressed Grid exceeds the Standard ILJK CG version for small system sizes. While on Test Platform 2 the Compressed Grid layout is able to raise the performance noticeable, it is doubtful, if the optimization effort of 3D-Blocking is worthwhile considering the small increase. Figure 5.8 and Table 5.1 show the best versions for the two Pentium 4 based platforms.



(a) Test Platform 1 (Xeon, 2.66 GHz)

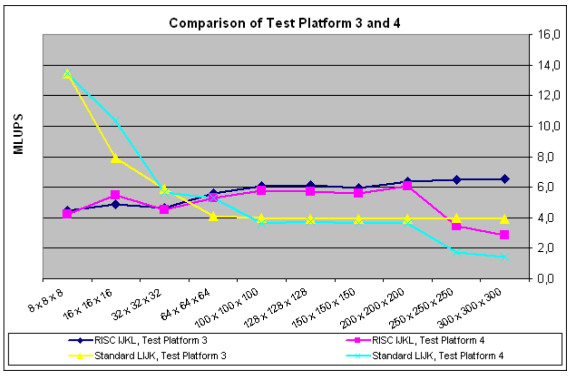


(b) Test Platform 2 (Pentium 4, 3.0 GHz)

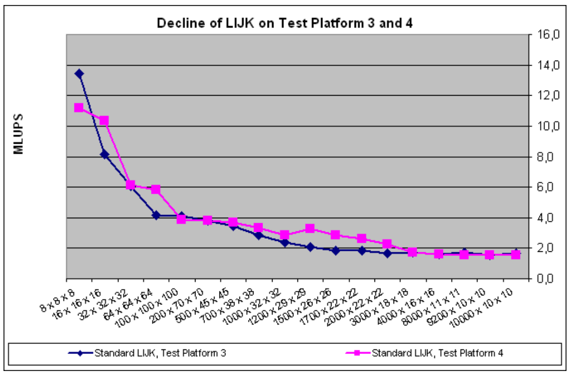
Figure 5.8: Best versions for Test Platforms 1 and 2.

Table 5.1: Overview of optimization techniques with best results for Test Platform 1 (Xeon, 2.66 GHz) and Test Platform 2 (Pentium 4, 3.0 GHz).

	best method				performance
	version	data layout	grid layout	blocking	(MLUPS)
Platform 1					
long systems	Standard	ILJK	Two Grid	no	3.3
cubic systems	Standard	ILJK	Two Grid	no	**2.9**
Platform 2					
long systems	Standard	ILJK	Compressed	no	4.1
cubic systems	Standard	ILJK	Compressed	no	**3.5**
	Standard	LIJK	Compressed	3D	**3.6**



(a) RISC IJKL and Standard LIJK



(b) Declines of LIJK

Figure 5.9: Performance comparison between Test Platform 3 (Itanium 2, 1.4 GHz) and Test Platform 4 (Itanium 2, 1.3 GHz).

5.1.3 Itanium 2

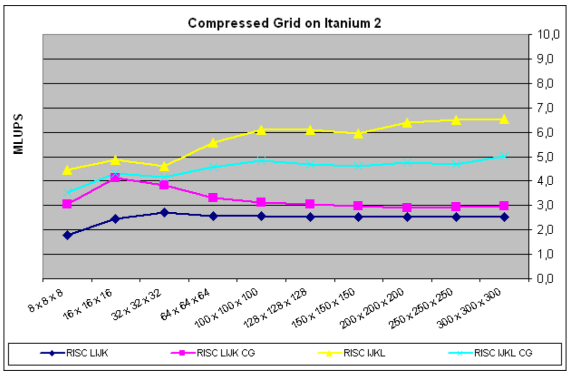
Comparing the two Itanium 2 based test platforms, it could be expected that the bigger cache of Test Platform 4 (3.0 MB instead of 1.5 MB) can raise performance more than the slower clock rate (1.3 GHz compared to 1.4 GHz) draws back. However, Figure 5.9(a) shows that only for the cache based system size 16^3 the bigger cache improves the performance, while on larger system sizes the results are equal. The decline of the LIJK version due to increasing dimension i, which depends on cache size (see Equation 5.1), are not as sharp as on other systems. The low slope is due to the higher associativity and larger capacity of the caches. The downward movement ends at the theoretical values of $1,700$ on Test Platform 3 (1.5 MB cache) and $3,500$ on Test Platform 4 (3.0 MB cache), as Figure 5.9(b) denotes.

The performance drop of the DSM architecture Test Platform 4 for huge systems larger than 250^3, shown in Figure 5.9(a), results from the lower interconnect bandwidth of the NUMAlink architecture compared to main memory bandwidth of the Itanium 2 processor. If the total memory consumption exceeds the local 8 GB, the CPU has to access memory of other nodes, which is more costly.

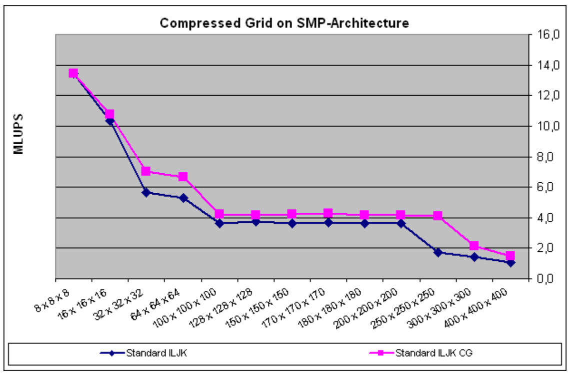
The Compressed Grid layout leads on the Itanium 2 based platforms to an improvement for LIJK data layout, only (about 15%). In contrast, there is a significant reduction for IJKL and ILJK data layouts (see Figure 5.10(a)). However, the DSM architecture Test Platform 4 profits from Compressed Grid on large system sizes. Due to the reduced memory consumption of Compressed Grid, the performance drop does not occur before 300^3 (see Figure 5.10(b)).

The Itanium 2 based platforms prove, that the assumption in section 4.3.1 is right: 1D-Blocking enhances LIJK versions only, while IJKL and ILJK data layout get worse. The shortening of dimension i cannot improve the performance of the RISC IJKL and ILJK versions, because the large caches of Itanium were no limit before (as in case of Pentium 4, for example). Figure 5.11 shows, that only the LIJK layout profits from 1D-Blocking, because the decline is replaced by a constant curve progression. Compressed Grid can only improve RISC LIJK slightly. Nevertheless, for long systems the best version is RISC IJKL.

While 3D-Blocking improved Standard LIJK by about 55%, on Itanium 2 RISC LIJK decreased



(a) Test Platform 3 (Itanium 2, 1.4 GHz)



(b) Test Platform 4 (Itanium 2, 1.3 GHz)

Figure 5.10: Compressed Grid layout on Itanium 2 based test platforms.

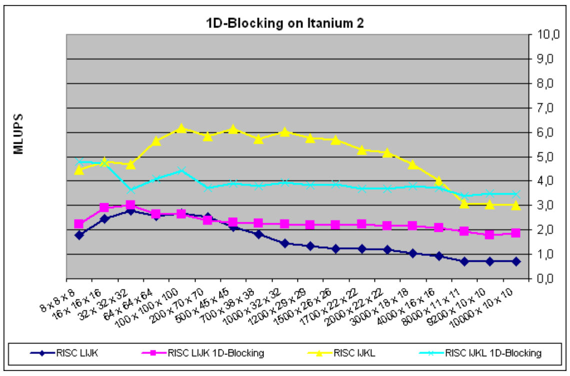
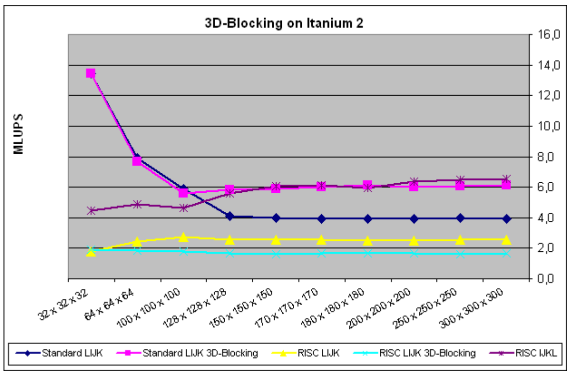
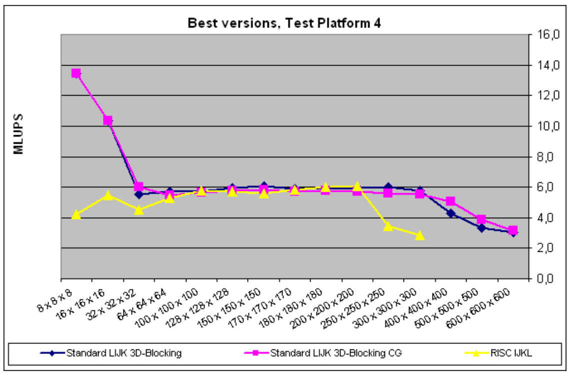


Figure 5.11: 1D-Blocking on Test Platform 3 (Itanium 2, 1.4 GHz).



(a) Test Platform 3 (Itanium 2, 1.4 GHz)



(b) Test Platform 4 (Itanium 2, 1.3 GHz)

Figure 5.12: 3D-Blocking in comparison to best versions on Itanium 2 based test platforms.

Table 5.2: Overview of optimization techniques with best results for Test Platform 3 (Itanium 2, 1.4 GHz) and Test Platform 4 (Itanium 2, 1.3 GHz).

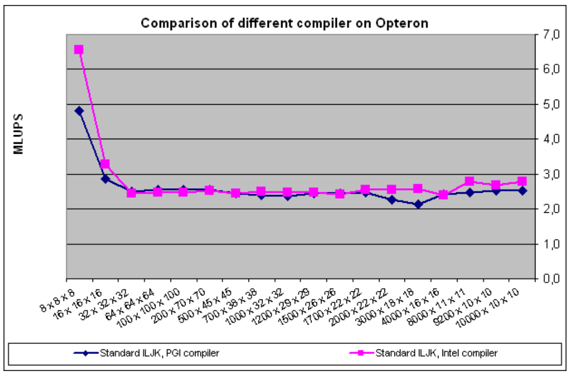
	best method				performance
	version	data layout	grid layout	blocking	(MLUPS)
Platform 3					
long systems	RISC	IJKL	Two Grid	no	5 - 6
cubic systems	RISC	IJKL	Two Grid	no	**6.5**
Platform 4					
long systems	RISC	IJKL	Two Grid	no	5 - 6
cubic systems	RISC	IJKL	Two Grid	no	**6.0**
huge systems	Standard	LIJK	Compressed	3D	3 - 5

in performance (see Figure 5.12(a)). It is assumed, that the Itanium 2 processor is very sensitive on short loops. The small block size results for RISC version in many short loops and cause this degradation in performance. On DSM architecture Test Platform 4 the 3D-Blocking technique significantly improved the performance on system sizes beyond the 8 GB. This indicates that 3D-Blocking in fact increase spatial (and temporal) locality, which obviously shows the largest improvement when the access of data is limited heavily by a bandwidth bottleneck. While the Compressed Grid layout does not improve 3D-blocked methods in general, the DSM architecture benefits from the reduced memory consumption (Figure 5.12(b)).

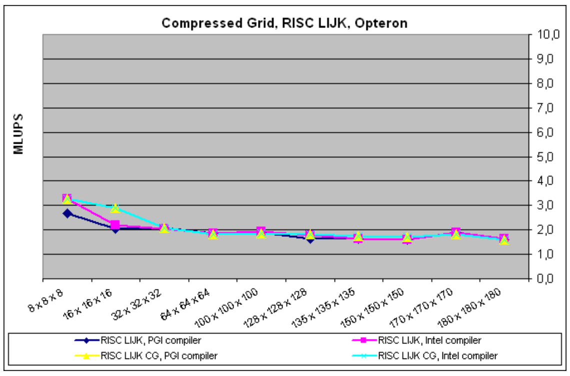
In conclusion, Table 5.2 presents the best versions for the Itanium based test platforms. While on Itanium 2 platforms in general RISC IJKL shows the best performance for both, long and cubic systems, on DSM-Test Platform 4 the optimization effort for 3D-Blocking and Compressed Grid is worthwhile when large systems beyond the local memory size are needed.

5.1.4 Opteron

The performances of *LBMKernel* compiled with Intel and Portland compiler do not differ seriously. Except on long systems (see Figure 5.13) and when applying 3D-Blocking (see Figure 5.15(b)), Intel compiler produces slightly better results, which is surprising because Intel IA32 compiler cannot use

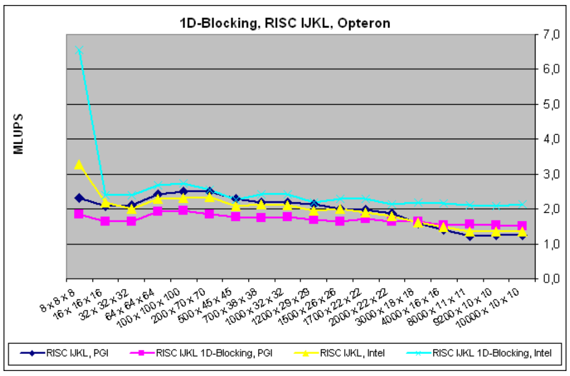


(a) Standard ILJK

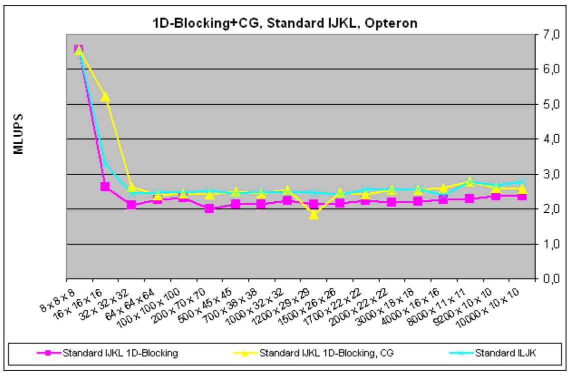


(b) Compressed Grid, RISC LIJK

Figure 5.13: Performance comparison between Intel and Portland compiler on Test Platform 5 (Opteron, 1.6 GHz).



(a) Different compilers for RISC IJKL 1D-Blocking



(b) 1D-Blocking with Intel compiler

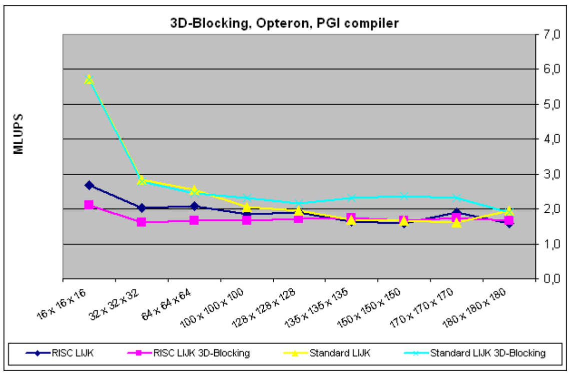
Figure 5.14: Performance inprovement of 1D-Blocking and Compressed Grid on Test Platform 5 (Opteron, 1.6 GHz).

all registers of the Opteron architecture.

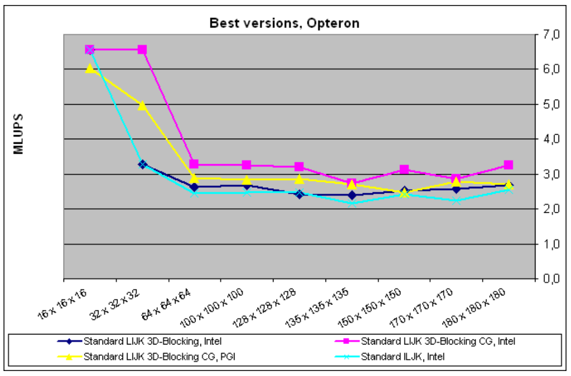
Using only the Compressed Grid layout takes no effect on Opteron with both Intel and Portland compiler (see Figure 5.13(b)). However, the combination with blocking techniques results in slight improvements (see later).

Although the effects of 1D-Blocking technique on Opteron mainly agree with the descriptions in section 5.1.1, there are few exceptions: While 1D-Blocking technique on Opteron could improve RISC IJKL and ILJK version, when using the Intel compiler, Portland's compiler could not achieve an enhancement (see Figure 5.14(a)). On Opteron platform, the Compressed Grid in combination with 1D-Blocking technique improves additionally. However, this combination results in a performance that is already achieved by Standard ILJK version (Figure 5.14(b)). Thus, for long systems it is not reasonable to perform 1D-Blocking and Compressed Grid optimization.

While the 3D-Blocking technique could enhance the Standard LIJK version for both Intel and PGI



(a) Increases and decreases by 3D-Blocking with LIJK data layout



(b) Best versions on Test Platform 5 (Opteron, 1.6 GHz)

Figure 5.15: 3D-Blocking and best versions on Test Platform 5 (Opteron, 1.6 GHz).

Table 5.3: Overview of optimization techniques with best results for Test Platform 5 (Opteron, 1.6 GHz) with Intel IA32 compiler and Portland 64-Bit compiler.

	best method				performance
	version	data layout	grid layout	blocking	(MLUPS)
Intel compiler					
long systems	Standard	ILJK	Two Grid	no	2.7
cubic systems	Standard	LIJK	Compressed	3D	**3.2**
Portland compiler					
long systems	Standard	ILJK	Two Grid	no	2.6
cubic systems	Standard	ILJK	Compressed	3D	**2.8**

compiler by about 40% to 50%, the RISC LIJK version is improved by Intel compiler only slightly, and got worse with PGI compiler (see Figure 5.15(a)). Despite to the other test platforms, on Opteron based Test Platform 5 the combination of Compressed Grid and 3D-Blocking technique resulted in an absolute maximum of performance (see Figure 5.15(b)). For both Intel and PGI compiler the 3D-blocked version of Standard LIJK with Compressed Grid is significantly the best. Here, the difference between results of Intel compiler and Portland compiler is extensive: Performance with Intel's IA32 compiler is roughly 15% better.

5.2 Profiling and Simulation

A common way to find reasons for good or bad performance is profiling, where performance influencing events are counted by using statistical methods or special hardware counters. The number and type of such hardware counters is architecture specific and sometimes not reliably due to bugs. Intel for example documented that the “2nd-level Cache Read Misses” event is afflicted by a bug that can cause both undercounting and overcounting by as much as a factor of 2 (see [Int99]). Unfortunately, the circumstances under which this bug occurs are not known, so that results cannot be used seriously. Also for “2nd-level Cache Load Misses Retired” event there is a notice that undercounting can occur, so that the results of this event has to be challenged, too. However, profiling can just give a result in terms of numbers, but no reason why the number is that high or low. Even with line based profiling it is impossible to determine exactly at which instructions the events occurred (due to the resolution of measurement). To evade these problems, in this thesis the cache simulating software *kcachegrind* was used [Wei03]. It examines the application program, and predicts cache behavior including assignment of cache miss causing instructions. Since this software simulates behavior of a cache structure with similar key aspects as Pentium 4 architecture has, the results can be partially included in investigation for Pentium 4 optimization. Unfortunately, although the results are very detailed and reasonable, the author of *kcachegrind* could not implement an exact and realistic imitation of a real architecture [Wei04]. Thus, results of *kcachegrind* can only be used as clues.

In the framework of this thesis, different tools for profiling were used. For a detailed description of these tools and their capabilities please refer to Appendix B.

5.2.1 Results of Profiling

In course of this thesis much time was spent on profiling, and much data was collected. In the following, only a small selection can be presented. The examples shown have been selected such that similarities as well as differences between the different architectures, algorithms and data layouts are demonstrated.

Profiling provides a better understanding why the performance of IJKL data layout compared to LIJK is better, and why the RISC version is slower than Standard. First, mispredicted branches have no significant impact since all versions have a prediction rate higher than 98%. Second, the CPI (cycles per instruction) ratio is much less for IJKL data layout than for LIJK. The main performance bottleneck are the cache misses: RISC version generally has three times more cache and DTLB misses than Standard version. The IJKL data layout causes only half of cache misses than LIJK data layout.

Figure 5.16 shows the most important performance limiting issues for the three versions, each in LIJK, IJKL and ILJK data layout without and with 1D-Blocking (except Vector version, which was not blocked and is not capable for ILJK data layout) on Test Platform 4 (Itanium 2). Unfortunately, since this platform is a DSM architecture, the failure of measurements concerning memory bandwidths is relatively high (e.g., especially the cases “RISC IJKL [1D-Blocking]” and “Std. LIJK [1D-Blocking]” are contradictory regarding L3 misses and performance). Comparison of the profiling results with measured performance (MLUPS values), shows clearly that the main performance limiting aspect are the L3 Cache Misses, while L2 Cache Misses are costly, but not as much, and DTLB misses do not influence the performance (since they nearly do not occur).

On Pentium 4 again the number of DTLB misses is small and cannot have severe impact on per-

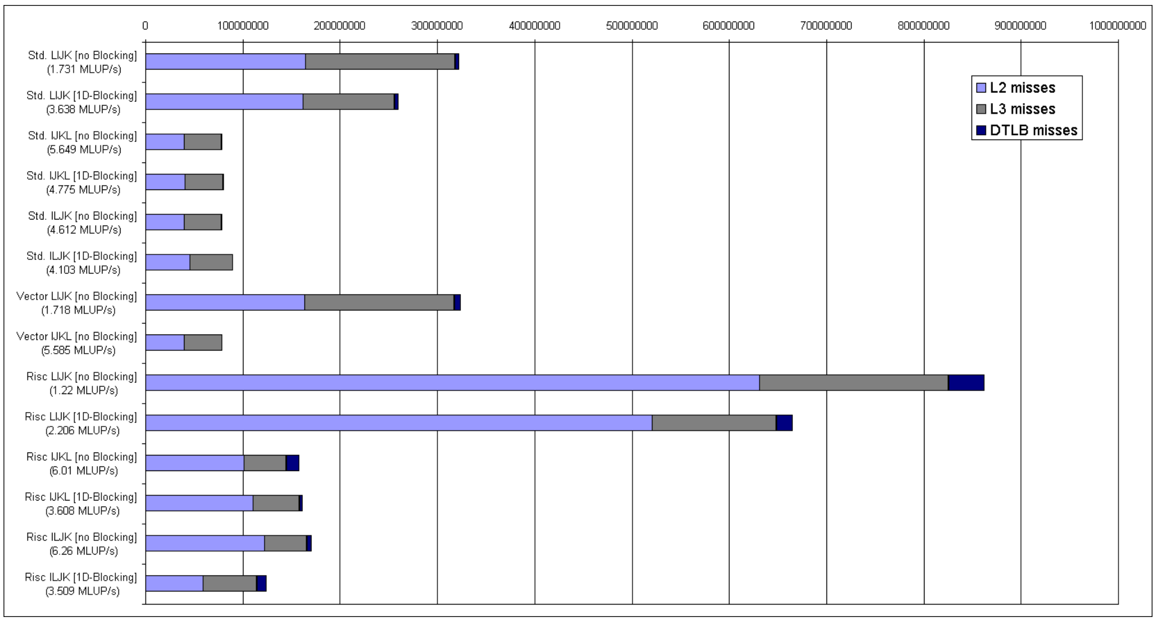


Figure 5.16: Comparison of Key Performance Limiting Events on Test Platform 4, using Two Grid layout and a system size of $3000 \cdot 18 \cdot 18$.

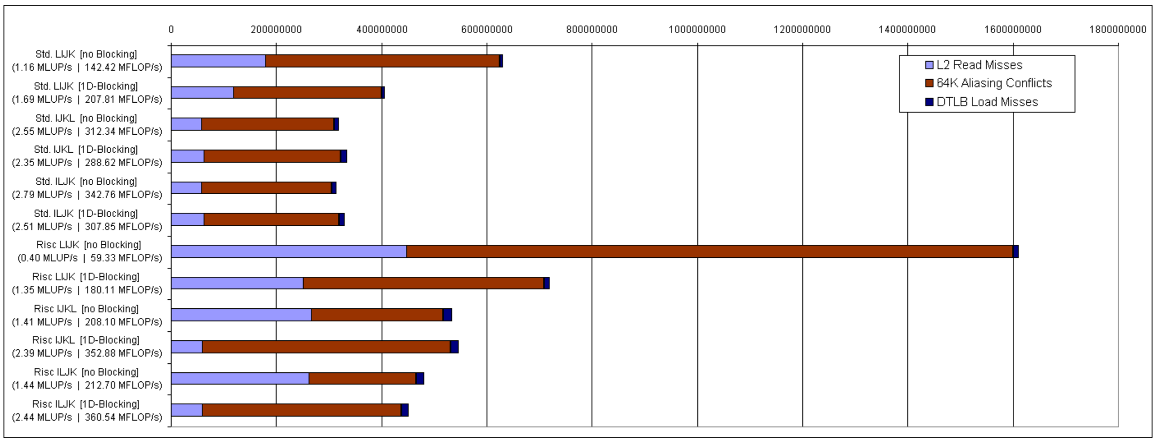


Figure 5.17: Comparison of Key Performance Limiting Events on a Pentium 4 based platform. Standard and RISC Version with Two Grid layout and a system size of $3000 \cdot 18 \cdot 18$.

formance. Figure 5.17 shows events, that explain the different performances: The 64K Aliasing Conflicts have a noteworthy impact on performance. Main performance limiting event surely are the L2 misses. It turned out, that the "2nd Level Cache Load Misses Retired" cannot explain the resulting performance as good as the "2nd Level Cache Read Misses", which are therefore used in this graph. Unfortunately, this event is subdued again by the Pentium 4 architecture bug (It is assumed that especially the RISC routines are overcounted). The differences of performance of Standard IJKL and Standard ILJK versions while having similar profiling results is probably due to measuring errors. The fact, that Standard LIJK is slower than RISC LIJK [1D-Blocking] while having better profiling results let assume that there is an additional event that influences performance on Pentium 4. The values for MFLOP/s result from profiling of events concerning floating point operations.

Further measurements showed, that the RISC version with IJKL layout is faster on a system size of 100^3 than on a system size of $3000 \cdot 18 \cdot 18$ because it produces more cache misses in latter case, even though the overall size of both systems is equal. This led to the idea of 1D-Blocking, which reduced cache misses and improved performance for the longer system. Reasons for the different number of cache misses are not known exactly. Since the 19 directions are not further apart in one of the two cases than in other, the only explanation could be that in case of the longer i-dimension more conflict misses with written cache lines occur (probably due to the longer temporary arrays). This issue led to the idea, that the 19 directions could be brought closer together without loosing the unit stride on i, which is realized by the ILJK data layout. In fact, this layout led to better performances on Pentium 4 platforms. On Itanium 2 there is no improvement, probably due to the larger and the higher set associative cache of Itanium 2 or Pentium 4's known problem with 64K Aliasing Conflicts that were reduced this way. Measurements affirmed that IJKL data layout and ILJK data layout have nearly same number of cache misses, but IJKL produces – depending on system size – more 64K aliasing conflicts and DTLB page walks (most probably due to the greater stride between 19 directions), which can be seen in Figure 5.17.

Table 5.4: Floating point operations per lattice site update as result from profiling on a Pentium 4 based platform, with Two Grid layout, no blocking.

	Standard			RISC		
	LIJK	IJKL	ILJK	LIJK	IJKL	ILJK
MFLOP/s	142.4	312.3	342.8	59.33	208.1	212.7
MLUP/s	1.16	2.55	2.79	0.40	1.41	1.44
Flops/LUP	122.9	122.7	122.7	148.0	147.9	147.9

From the values of MLUP/s and MFLOP/s in Figure 5.17, the number of floating point operations that are required to update one lattice site can be computed. The result is shown in Table 5.4. It can be seen that floating point operations per lattice site update ratio (Flops/LUP) only depends on the used version, but not on data layout. Up to now there is no explanation why the number of Flops differs so much between Standard and RISC version, as the number of Flops in the code is equal. The measured values surely are smaller than the theoretically counted 156 since the compiler can optimize by using architecture specific instructions like the fused multiply-add or SSE-instructions.

Since write misses in write-back caches are very costly, it was tried to measure them by profiling. Unfortunately Intel's architectures do not support a separate counter for write misses. According to the *VTune* reference [Int00] which defines two different counters regarding L2 cache misses on Pentium 4, an estimation for write misses may be calculated: While the "2nd-level Cache Load Misses Retired" counts the number of load instructions, that attempt to load data from L2 cache, but the data was not in cache (thus, a L2 cache read miss occurs), the event "2nd-level Cache Read Misses" is defined as "The number of 2nd-level cache read misses as seen by the bus queue for memory load misses and RFO misses". The latter includes additionally those cache misses that were caused initially by a write access to cache, which caused a read miss (due to write-allocate). Therefore, the difference of both events is regarded to be a measure for cache write misses. Thus, as an estimation for algorithm performance, the miss per lattice site update (Miss/LUP) ratio were measured and calculated. That is the number of highest level cache misses per lattice site update in mean. After subtraction of misses caused by initialization of the array, the ratio only depends mainly on the version, data layout and memory strategy used (see Table 5.5).

Table 5.5: Read and write misses per lattice site update resulting from profiled data for nonblocked case on Two Grid layout on a Pentium 4 based platform. Values of write misses for RISC versions seem to underlie the failure of factor 2 in "2nd Level Cache Read Misses" event. Values in paranthesis are theoretic simulation results from kcachegrind.

	Std. LIJK	RISC LIJK	Std. IJKL	RISC IJKL	Std. ILJK	RISC ILJK
Read misses/LUP	2.7 (2.3)	12.6 (2.5)	1.7 (2.4)	1.9 (2.5)	1.8 (2.4)	2.4 (2.5)
Write misses/LUP	3.3 (4.8)	N/A (4.8)	2.1 (2.3)	0.9 (2.3)	1.5 (2.3)	0.6 (2.3)

Taking into account the assumptions of section 4.1, Table 5.5 shows the measured values of misses per lattice site update. For IJKL data layout in Standard version, for example, there was predicted, that in mean each 16th loop 19 cache read misses and 19 cache write misses occur. This would result in 1.2 read misses per lattice site update (LUP) and the same amount for write misses per LUP. The table shows, that the routines in reality have more read misses. The values for write misses/LUP are again bound to the Pentium 4 bug. Most probably this is the reason why for RISC LIJK version no meaningful values are determinable. The more theoretic simulation results of *kcachegrind* are not reasonable, too (values in paranthesis in Table 5.5). Despite they show the same magnitude of read and write misses for IJKL and ILJK data layout and more write misses than read misses for LIJK data layout (as predicted), no significant differences between Standard and RISC version is noticeable. Surely the reason for that is, that *kcachegrind* does not take the `VECTOR NONTEMPORAL` directive into account and cannot simulate the bottleneck of memory bandwidth realistically, which is due to the simulated cache structure (see section B.2 for more details).

5.2.2 Simulation with kcachegrind

Generally speaking, the results of simulation agree well with the theoretical assumptions. The distribution of cache read misses and cache write misses match to the assumptions made in section 4.1. Unfortunately, simulation differs from most of profiling results, especially in those cases, where profiling is afflicted by Pentium architecture's bugs. So *kcachegrind* confirmed the assumption, that IJKL and ILJK data layout produces write misses that are distributed uniformly on all 19 write back instructions (for both Standard and RISC version). Same is true for reading values. The LIJK data layout shows irregular appearance of cache read and write misses. This is due to the relative displacement of 19 `doubles` in the 16 `doubles` long cache lines. However, despite the cache space wasting temporary arrays of RISC version *kcachegrind* assumes same magnitude of cache misses for different algorithm versions (see Table 5.5) which coincides neither with profiling results nor with theoretical suppositions.

The comparison of Standard IJKL and RISC IJKL shows, that RISC version has more read misses than Standard version (where the magnitude of difference seems to be too small). As mentioned already before, Standard version has better chances to hold data from preceding iteration in cache, which saturates next iterations, too, due to the unit stride on i, while in contrast RISC version's long temporary arrays waste much cache space. When using Compressed Grid, the amount of write misses is nearly same. The RISC version performs much more instructions for reading from and storing to temporary fields, the more frequent evaluation of `obstacleField` and the additional overhead due to the higher number of loops.

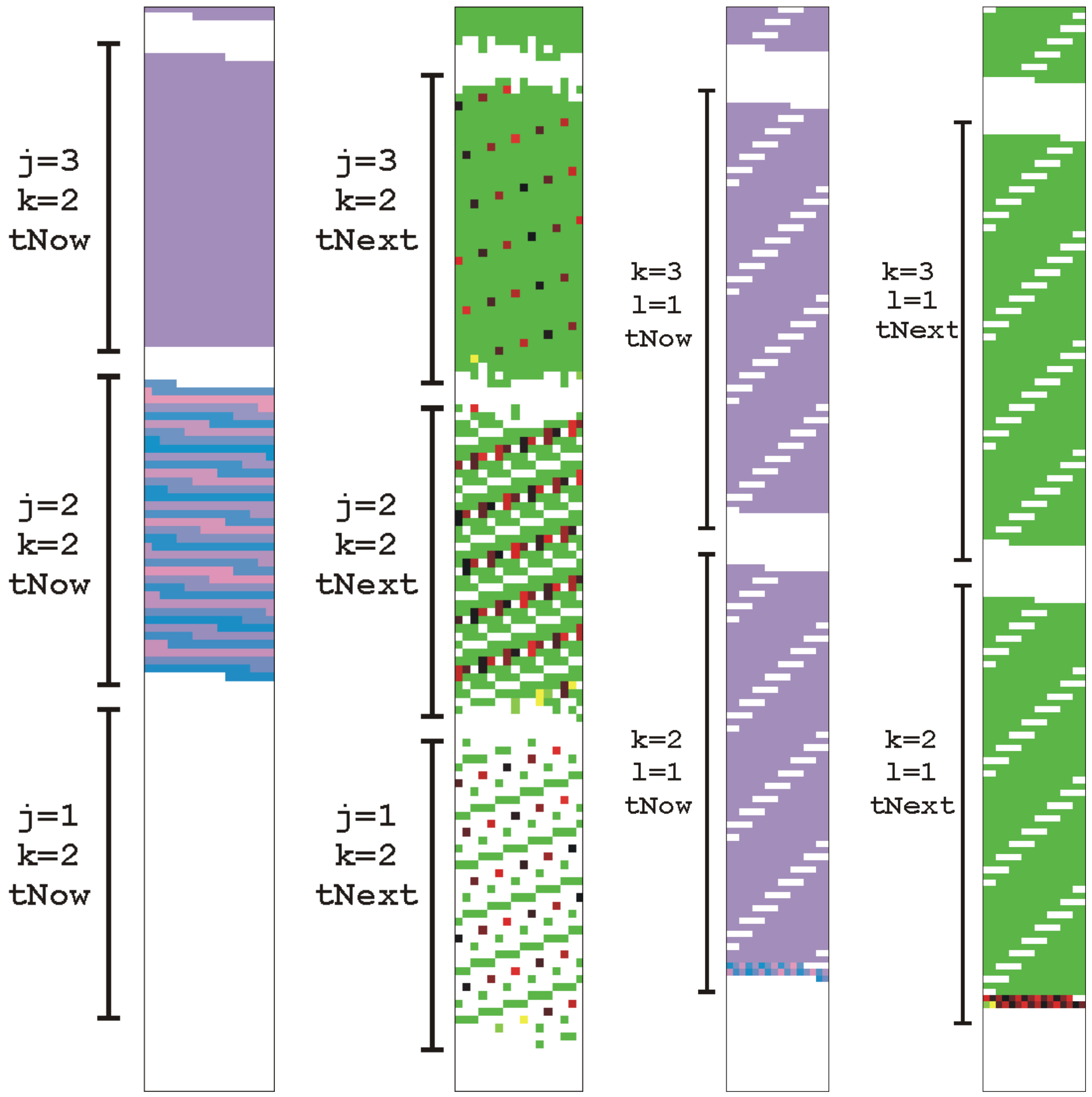
Comparing Standard LIJK and RISC LIJK, the simulation approves that in each iteration the first access on `pdf` causes a cache miss and in the following only few read misses occur. This is because of the subsequent order of the 19 directions. Here again, RISC version has much more instructions for writing updated values.

5.3 Memory Visualizer MemPHis

While *kcachegrind* is able to give detailed information which instruction cause cache misses, it is also interesting to know, in which order these memory accesses occurred. For this purpose a memory visualizer, called *MemPHis*, was developed within the framework of this thesis. The results given by *MemPHis* library were a good graphical aid for developing better ideas or reasonings while profiling delivered the facts in numbers. The examination of different output bitmaps for IJKL data layout helped in raising the suspicion that the wide stride for the 19 velocities should be shortened wherefore the ILJK data layout was developed. For details on source code and implementation of *MemPHis* see Appendix C.

Figure 5.18 shows exemplary visualization outputs of different memory access patterns. These figures were produced with Standard version, using a system size of 32^3. The width of the pictures is 16, which is the length of a cache line in `doubles`. This helps for estimation which memory accesses caused cache misses. In Figure 5.18(a) the reading access on LIJK data layout with Two Grid layout is shown. In row with index $k = 2$ and $j = 2$ for every iteration of i an other color was used. Thus, the 30 sets of 19 directions are well identifiable. Between two layers the white spaces represent the two rows of boundary cells and two rows of ghost cells. Under the assumption, that `pdf` array is aligned, thus, starting on a memory address that maps to the beginning of a cache line, the displacement of the 19 directions of adjacent cells in cache is in evidence. While image shows only two, in fact there are 30 such blocks. Separated by four white blocks (boundary and ghost rows), there are 30 such sets consisting of 30 blocks. Figure 5.18(b) shows the according layer in destination grid. The scattered pixels in lower region of image are values that were written into the boundary cells. The middle region represents the row with $k = 2$ and $j = 2$. All pixels with other color than green were written from updating process of lattice sites in the mentioned row.

Figure 5.18(c) shows read access using the IJKL data layout. Again, each of the cells in row with $k = 2$ and $j = 2$ received an other color. Here, one can see the 30 adjacent cells of one row lying consecutively in memory. Four white pixels in between represent again boundaries and ghost cells. These blocks, each of them consisting of 30 rows, are separated by white spaces that represent two boundary and two ghost rows. With 28 additional following blocks, they represent one whole grid, of which there are 19 in consecutive manner in the bitmap file, followed by 18 sets representing the target directions of second grid. Figure 5.18(d) shows the beginning of one target set, where the values of updated cells, that belong to row with $k = 2$ and $j = 2$, are written to.



(a) read LIJK layout (b) write LIJK layout (c) read IJKL layout (d) write IJKL layout

Figure 5.18: Cutouts of resulting images for Standard version with MemPHis on a 32^3 system.

5.4 Compiler Optimization

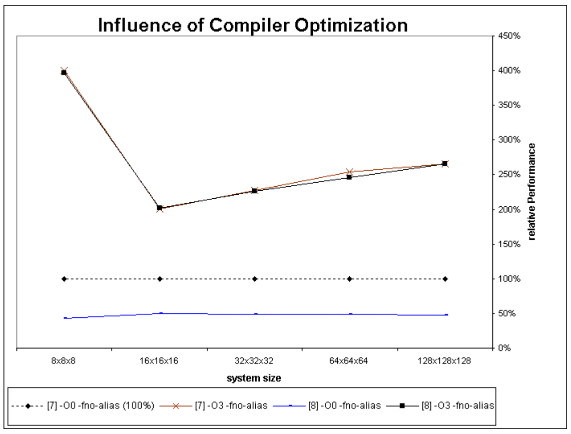
This section presents the impact of compiler flags and directives on performance of *LBMKernel*. In summary: most flags beyond the standard flag for high optimization (e.g. `-O3`) and directives provide only minor improvements. While it is obvious to choose highest optimization level, the difference between compiler builds and versions for IA32 compilers are not as significant as for IA64 compilers. Therefore, all measurements for optimization techniques were performed always with newest compiler and all possible optimization flags.

5.4.1 Intel IA32 Compiler

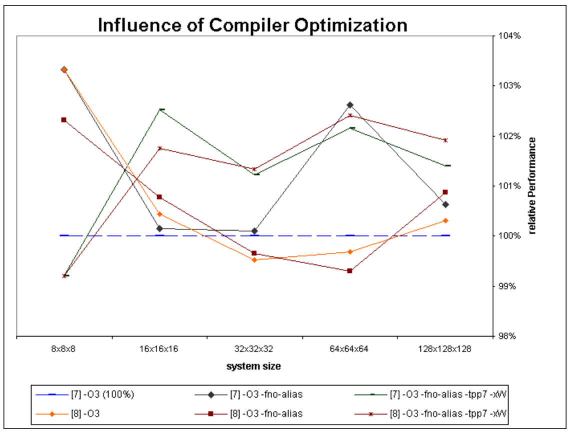
On Pentium 4 based Test Platform 2, only the optimization flag `-O3` shows a significant increase of performance by factor 2.5 (see Table 5.6 and Figure 5.19) compared to `-O0`. Comparison of different compiler versions shows that version 7.1 produce better performance when no optimization is activated, but same results for optimization activated. The flags for specifying Pentium 4 architecture (`-tpp7`) and that for specifying Pentium 4 compatible architecture (`-xW`) deliver no difference in performance. The `-fno-alias` flag does not increase performance. Examining the effect of directives for RISC version of *LBMKernel* (only there are these directives implemented) show no significant improvement (see Table 5.7 and Figure 5.20). The compiler in version 8.0 seems to produce better code with `VECTOR NONTEMPORAL` directive than compiler 7.1. But since the directives `!DEC$ VECTOR NONTEMPORAL` and `!DEC$ IVDEP` give no great improvement in comparison to the case where no directives were used, it only can be stated that compiler 8.0 creates better code for RISC version, in general. The directive `!DEC$ VECTOR ALIGNED` seems to have potential to improve performance in some cases (dependent on system size).

Table 5.6: Relative comparison of resulting performance for Standard IJKL version of LBMKernel for different system sizes. Measurements were taken with Intel's IA32 compilers in version 7.1 and 8.0, on Test Platform 2, with activated `VECTOR ALIGNED`, `VECTOR NONTEMPORAL` and `IVDEP` directives.

Compiler/Flags	128x128x128	64x64x64	32x32x32	16x16x16	8x8x8
[7] `-O0`	101 %	100 %	99 %	98 %	96 %
[7] `-O0 -fno-alias` (reference)	100 %	100 %	100 %	100 %	100 %
[7] `-O3`	264 %	247 %	227 %	200 %	387 %
[7] `-O3 -fno-alias`	265 %	254 %	227 %	201 %	400 %
[7] `-O3 -fno-alias -xW`	269 %	256 %	228 %	201 %	439 %
[7] `-O3 -fno-alias -tpp7`	262 %	251 %	226 %	201 %	341 %
[7] `-O3 -fno-alias -tpp7 -xW`	267 %	252 %	230 %	205 %	384 %
[8] `-O0`	45 %	45 %	46 %	47 %	40 %
[8] `-O0 -fno-alias`	47 %	48 %	49 %	50 %	43 %
[8] `-O3`	265 %	246 %	226 %	201 %	400 %
[8] `-O3 -fno-alias`	266 %	245 %	226 %	202 %	396 %
[8] `-O3 -fno-alias -xW`	275 %	241 %	232 %	202 %	421 %
[8] `-O3 -fno-alias -tpp7`	269 %	257 %	229 %	204 %	399 %
[8] `-O3 -fno-alias -tpp7 -xW`	269 %	253 %	230 %	204 %	384 %



(a) Performance difference for flags `-O0` and `-O3`.



(b) Performance difference for flags `-fno-alias` and `-xW -tpp7`.

Figure 5.19: Compiler influenced performance difference on Pentium 4 (Test Platform 2) for Intel IA32 compiler (Versions 7 and 8), measured with Standard IJKL version.

Table 5.7: Relative comparison of resulting performance for RISC IJKL version of LBMKernel for different system sizes. Measurements were taken with Intel's IA32 compilers in version 7.1 and 8.0, on Test Platform 2, with flags `-O3 -xW -tpp7 -fno-alias`. The abbreviation "NT" stands for `VECTOR NONTEMPORAL` directive.

Compiler/Flags	200x70x70	100x100x100	64x64x64	32x32x32	16x16x16	8x8x8
[7] `IVDEP` (ref.)	100 %	100 %	100 %	100 %	100 %	100 %
[7] `NT`	100 %	100 %	100 %	99 %	101 %	107 %
[7] `NT, IVDEP`	100 %	100 %	98 %	100 %	88 %	107 %
[8]	115 %	108 %	106 %	101 %	100 %	106 %
[8] `IVDEP`	116 %	107 %	105 %	100 %	100 %	103 %
[8] `NT`	116 %	108 %	105 %	101 %	99 %	105 %
[8] `NT, IVDEP, ALIGNED`	112 %	107 %	107 %	103 %	103 %	113 %

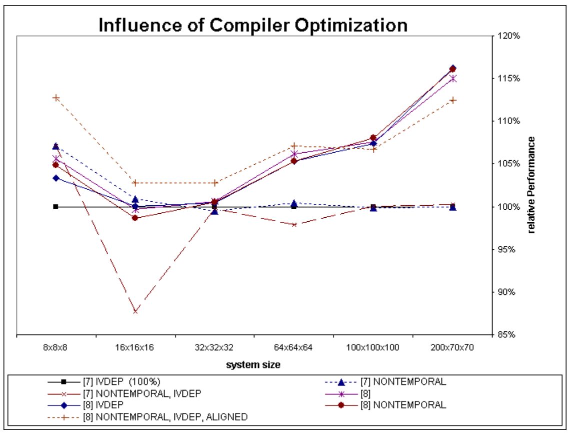


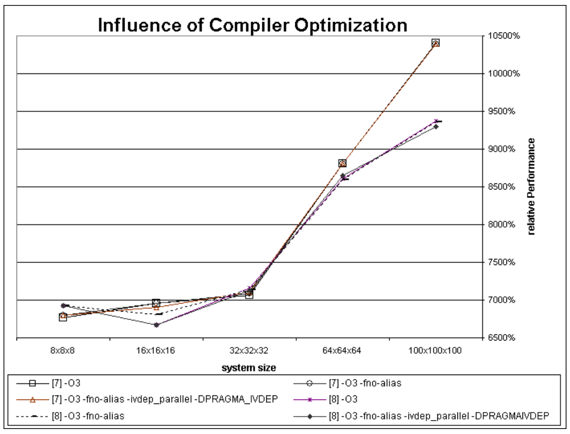
Figure 5.20: Performance Difference for `IVDEP`, `VECTOR NONTEMPORAL` and `VECTOR ALIGNED` directive for Intel IA32 compiler (Versions 7 and 8), measured on Test Platform 2 with RISC IJKL version.

5.4.2 Intel IA64 Compiler

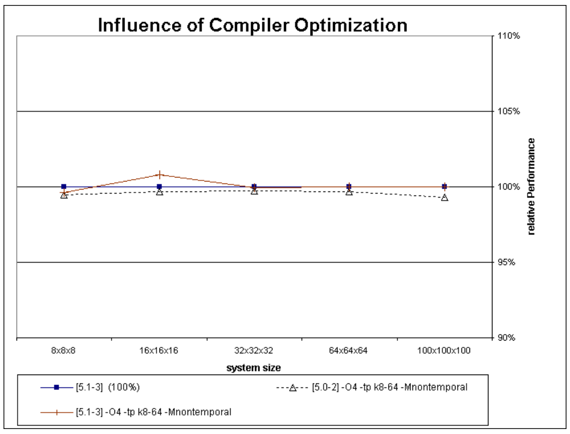
The resulting performance differences of flags and directives when using Intel's IA64 compiler are more significant. Measurements on Itanium 2 based Test Platform 3 (see Table 5.8 and Figure 5.21(a)) indicate that the compiler in version 7.1 generates best code with `-ivdep_parallel`. Similar to the results for IA32 compiler, here the version 7.1 produces better code than version 8.0 when optimization is disabled. But ratio of performance of optimized code to unoptimized version is significantly higher (about 100), which surely is due to the *EPIC* (Explicitly Parallel Instruction Computing) architecture of the Itanium 2 processor. In contrast to the out-of-order execution techniques of standard RISC processors, the compiler builds bundles which instructions to be executed in parallel. The consequence is, that code scheduling and thus performance is determined by the compiler only and no out-of-order support is available on the Itanium processor itself.

Table 5.8: Relative comparison of resulting performance for RISC IJKL version of LBMKernel for different system sizes. Measurements were taken with Intel's IA64 compilers in version 7.1 and 8.0, on Test Platform 3.

Compiler/Flags	100^3	64^3	32^3	16^3	8^3
[7] `-O0`	142 %	142 %	142 %	142 %	145 %
[7] `-O0 -fno-alias`	142 %	142 %	142 %	144 %	145 %
[7] `-O3`	10406 %	8812 %	7065 %	6958 %	6864 %
[7] `-O3 -fno-alias`	10392 %	8812 %	7065 %	6958 %	6904 %
[7] `-O3 -ivdep_parallel -fno-alias`	10392 %	8812 %	7105 %	6908 %	6904 %
[8] `-O0` (reference)	100 %	100 %	100 %	100 %	100 %
[8] `-O0 -fno-alias`	100 %	100 %	100 %	100 %	101 %
[8] `-O3`	9377 %	8605 %	7157 %	6667 %	7028 %
[8] `-O3 -fno-alias`	9366 %	8592 %	7137 %	6809 %	7028 %
[8] `-O3 -ivdep_parallel -fno-alias`	9300 %	8652 %	7131 %	6667 %	7028 %



(a) Performance difference for `-O0`, `-O3` and `-ivdep_parallel` with `IVDEP` directive. Regards Intel IA64 compiler (Versions 7 and 8). Measured on Test Platform 3 with RISC IJKL version.



(b) Performance difference for Portland's PGI compiler (Versions 5.0-2 and 5.1-3). Measured on Test Platform 5 with Standard IJKL version.

Figure 5.21: Compiler influenced performance difference on 64-Bit systems.

5.4.3 PGI Compiler 64-Bit

For Portland's PGI compiler only a comparison between the two versions and between using and not using of `-Mnontemporal` for version 5.1-3 were performed. Since the default optimization level is `-O1`, if `-O` is not specified, it is surprising that specifying `-O4` delivers no better performance. Efforts to find flags to sway the compiler producing codes with better performance led to no success (see Table 5.9 and Figure 5.21(b)).

Table 5.9: Relative comparison of resulting performance for Standard IJKL version of LBMKernel for different system sizes. Measurements were taken with Portland's PGI compilers in version 5.0-2 and 5.1-3, on Test Platform 5.

Compiler/Flags	100x100x100	64x64x64	32x32x32	16x16x16	8x8x8
[5.1-3] (reference)	100 %	100 %	100 %	100 %	100 %
[5.0-2] `-O4 -Mnontemporal`	99 %	100 %	100 %	100 %	99 %
[5.1-3] `-O4 -Mnontemporal`	100 %	100 %	100 %	101 %	100 %

Chapter 6

Conclusion

In the past decade the Lattice Boltzmann method has become an important method for calculating fluid dynamics. The method itself is memory intensive and due to its simplicity it provides a good testbed for performance optimization through improved data layout and memory access strategies. Thus, the focus of this thesis is to investigate the interplay of memory hierarchies and data access patterns on a wide range of modern processor architectures.

The common sequence for optimization of codes was followed: After the best available architectures were chosen, the best alternative of algorithm was selected. Three different solutions designed for different architectural key aspects have been implemented: Two versions for cache based systems and one version for vector architectures. Since scalar architectures have highly complex processor architectures, basic optimizations, e.g. instruction scheduling, have to be done by the compiler and thus, the best choice of compiler flags and compiler directives is important. For memory intensive codes like the Lattice Boltzmann method the most important part in the optimization process are memory related optimizations, e.g. definition of the data layout, which cannot be done by the compiler. Improving the spatial and temporal locality have been investigated by applying Compressed Grid technique as well as using different array layouts for the particle density distribution. Furthermore, efforts in common optimization tasks, like Line Blocking and Cubic Blocking were performed. The search for best performance was supported by third party profiling and simulation tools as well as the memory visualizer developed in the framework of this thesis.

A first result is, that the standard flag for highest optimization provides reasonable performance and subtle compiler options and directives only provide minor benefits. Of course, the architecture dependent choice of the implementation is important. However, the layout of data in memory has the highest impact on performance and must be adapted to the underlying architecture. In general, the correct choice of the data layout renders further complex optimization techniques like Line Blocking, Cubic Blocking dispensable. Moreover in this thesis each improve or decrease of performance could be reasoned by interpreting of profiling and simulation data in combination with architectural design details.

However, there is still potential for further performance improvements. Additional temporal blocking as done in [Igl03], e.g., resulted in further performance increase, but the impact on the data layout was not considered. For 3D-Blocking a detailed research of finding best suitable blocking parameters for the three dimensions in accordance to cache and memory layout may result in a better optimization strategy. As a further prospect, there still is the possibility of parallelization, since parallelizing Lattice Boltzmann method is not too difficult. Basic attempts in use of OpenMP were performed successfully in [Igl03], and detailed research in MPI was done by [PTD04].

Appendix A

Collection of Charts

For the sake of completeness below the performance results for each combination of optimization techniques and test platforms are presented. Starting with the intuitive Two Grid layout, all data layouts will be shown for the Standard version, Vector and RISC version (while ILJK data layout and Compressed Grid technique for Vector version is not applicable, and blocking was not implemented for Vector version), grouped by the test platforms. For better comparison with results of 1D-Blocking and 3D-Blocking, for each method without blocking two charts are presented. One, that draws the graph over the system sizes which were used for 3D Blocking (with cubic sizes), the other over the system sizes of 1D Blocking: From 8^3 to 100^3 the system size is increased in cubic way, above 100^3 the overall system size stays constant while enlarging dimension i and decreasing the other two dimensions accordingly.

A.1 Test Platform 1 (Xeon, 2.66 GHz)

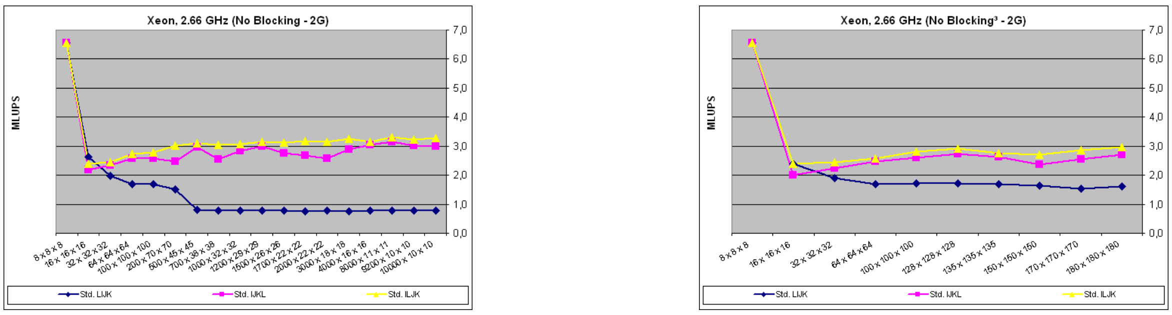


Figure A.1: Two Grid layout on Test Platform 1 (Xeon, 2.66 GHz), Standard version.

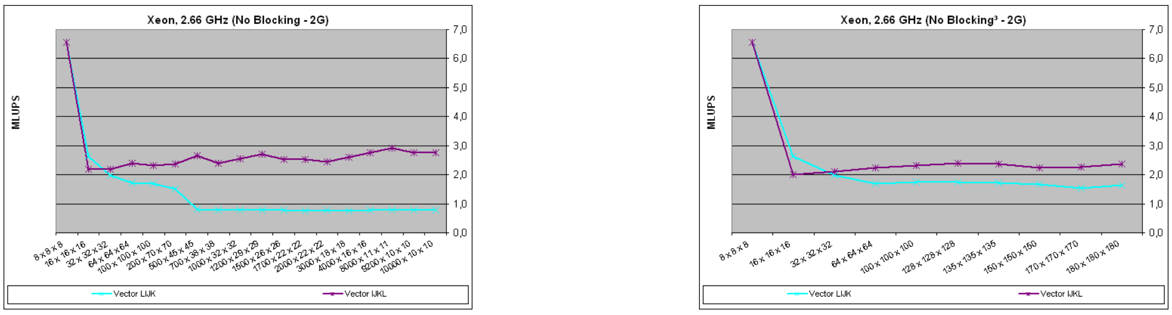


Figure A.2: Two Grid layout on Test Platform 1 (Xeon, 2.66 GHz), Vector version.

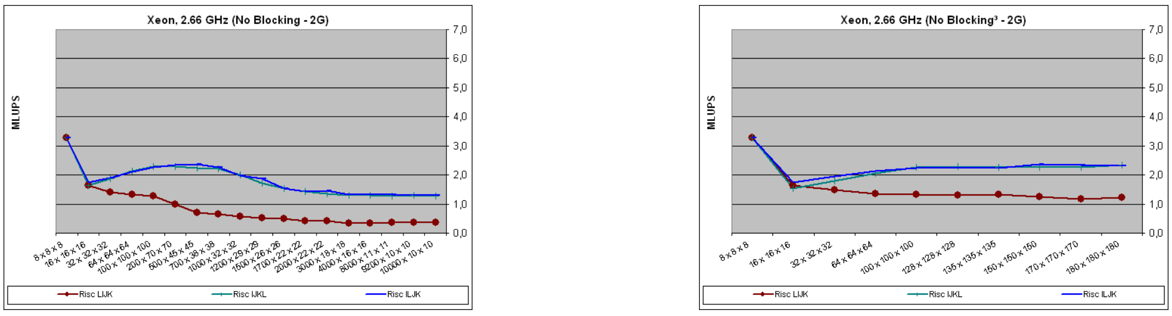


Figure A.3: Two Grid layout on Test Platform 1 (Xeon, 2.66 GHz), RISC version.

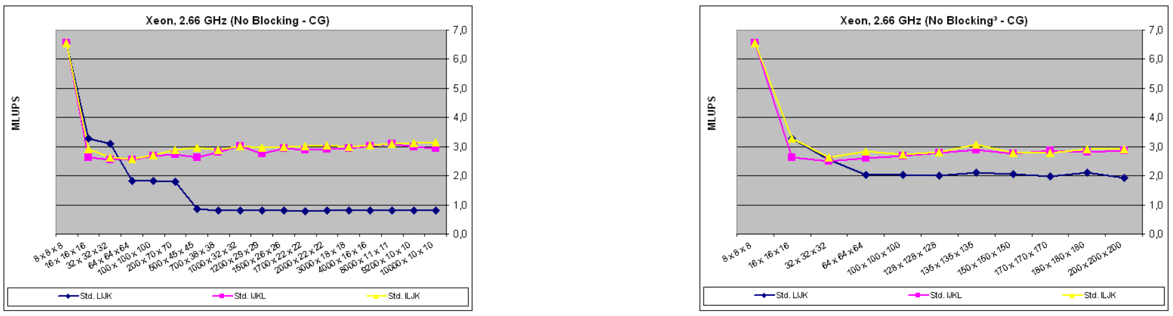


Figure A.4: Compressed Grid layout on Test Platform 1 (Xeon, 2.66 GHz), Standard version.

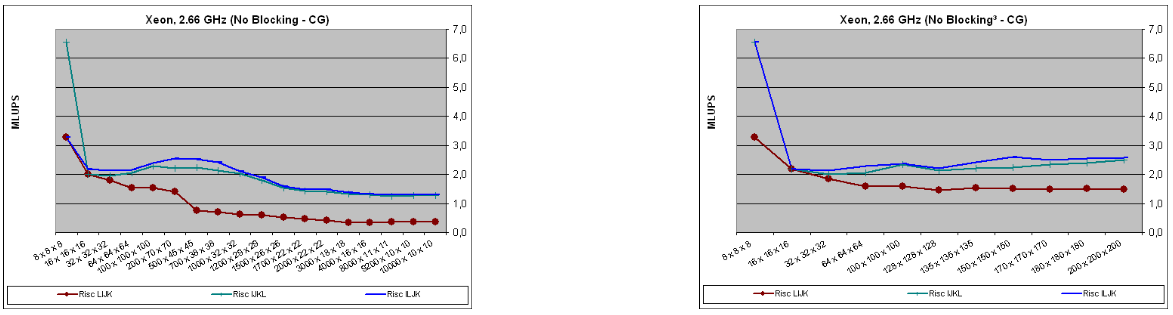


Figure A.5: Compressed Grid layout on Test Platform 1 (Xeon, 2.66 GHz), RISC version.

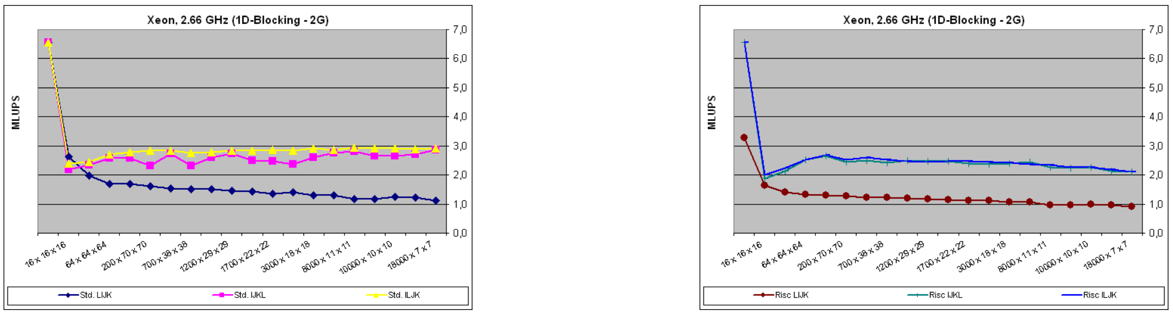


Figure A.6: 1D-Blocking, Two Grid layout on Test Platform 1 (Xeon, 2.66 GHz).

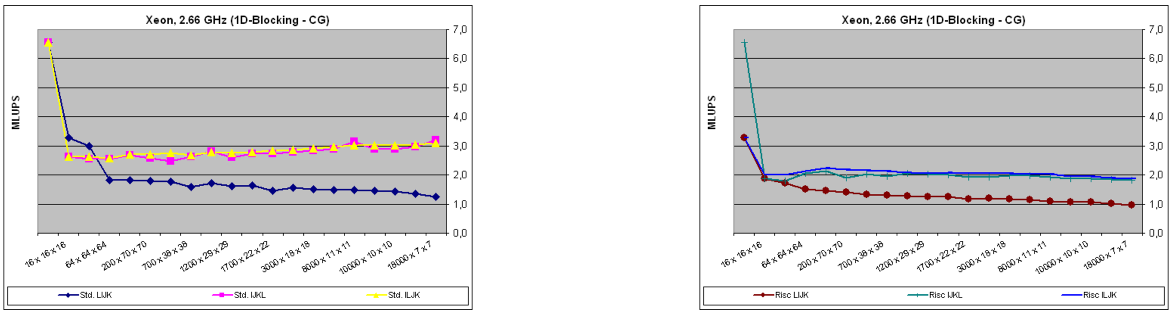


Figure A.7: 1D-Blocking, Compressed Grid layout on Test Platform 1 (Xeon, 2.66 GHz).

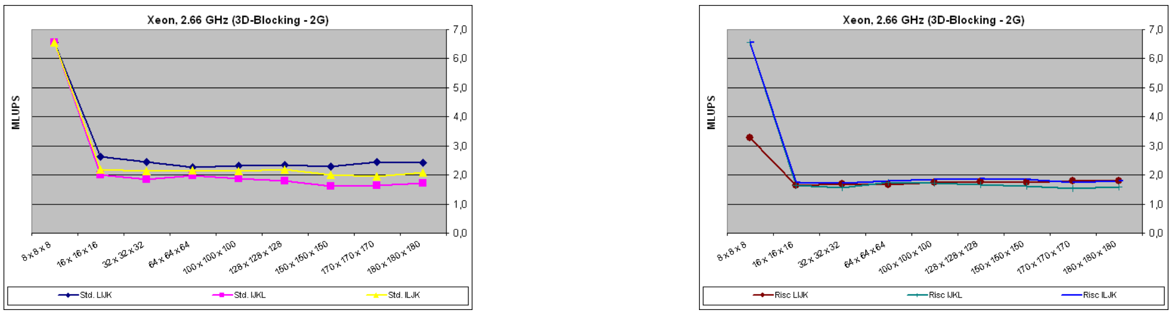


Figure A.8: 3D-Blocking, Two Grid layout on Test Platform 1 (Xeon, 2.66 GHz).

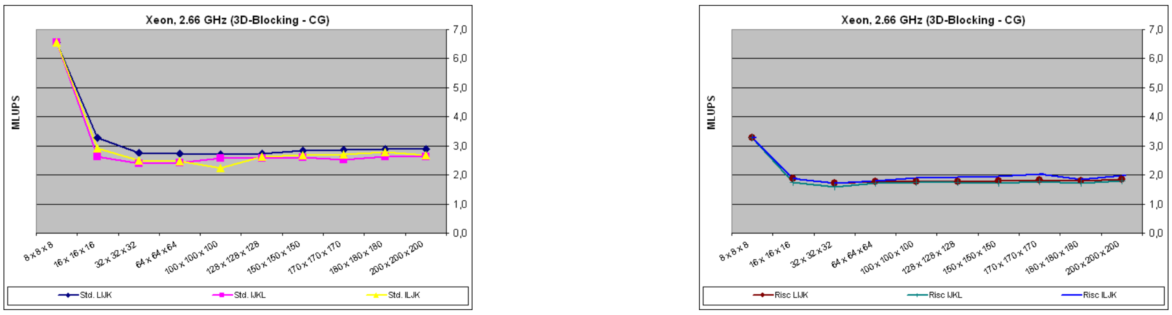
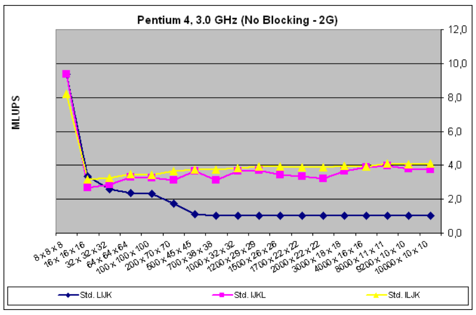


Figure A.9: 3D-Blocking, Compressed Grid layout on Test Platform 1 (Xeon, 2.66 GHz).

A.2 Test Platform 2 (Pentium 4, 3.0 GHz)



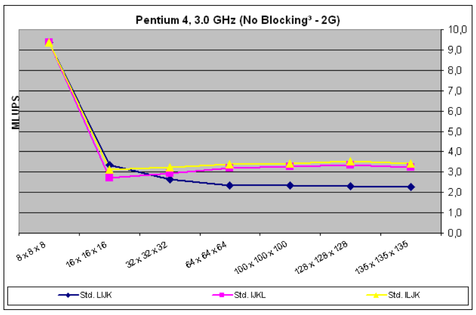
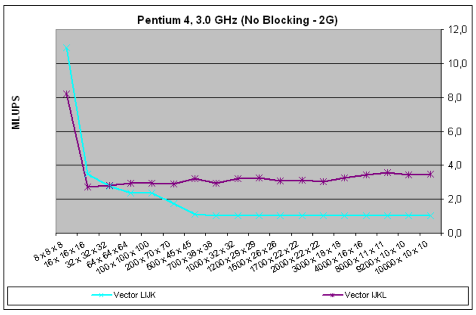


Figure A.10: Two Grid layout on Test Platform 2 (P4, 3.0 GHz), Standard version.



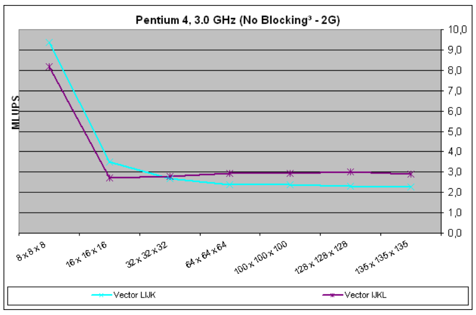
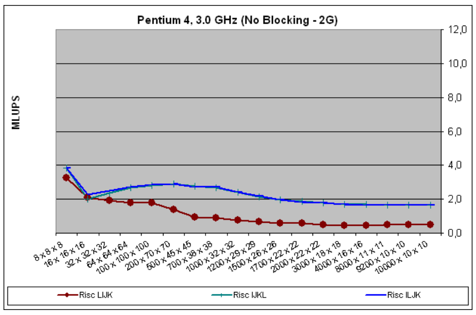


Figure A.11: Two Grid layout on Test Platform 2 (P4, 3.0 GHz), Vector version.



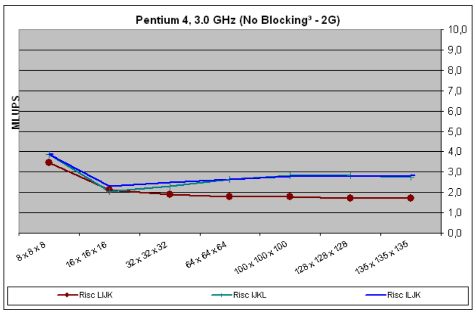


Figure A.12: Two Grid layout on Test Platform 2 (P4, 3.0 GHz), RISC version.

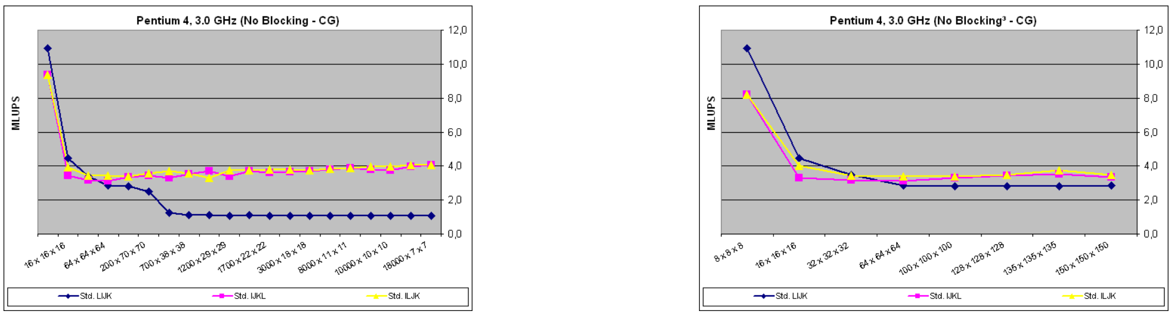


Figure A.13: Compressed Grid layout on Test Platform 2 (P4, 3.0 GHz), Standard version.

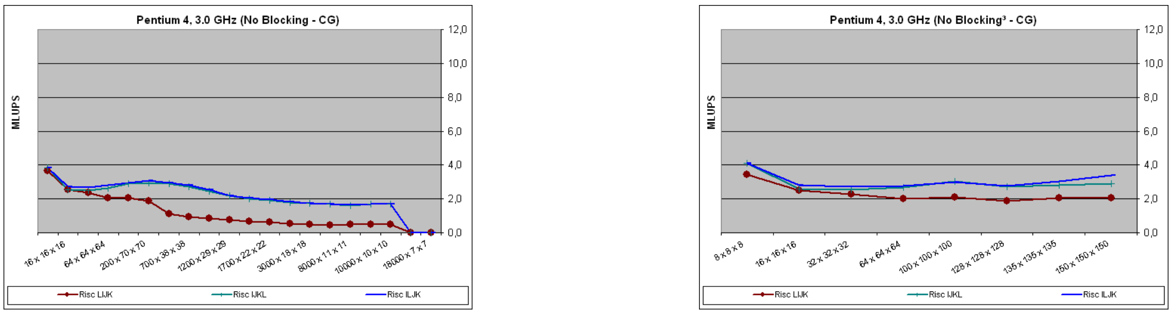


Figure A.14: Compressed Grid layout on Test Platform 2 (P4, 3.0 GHz), RISC version.

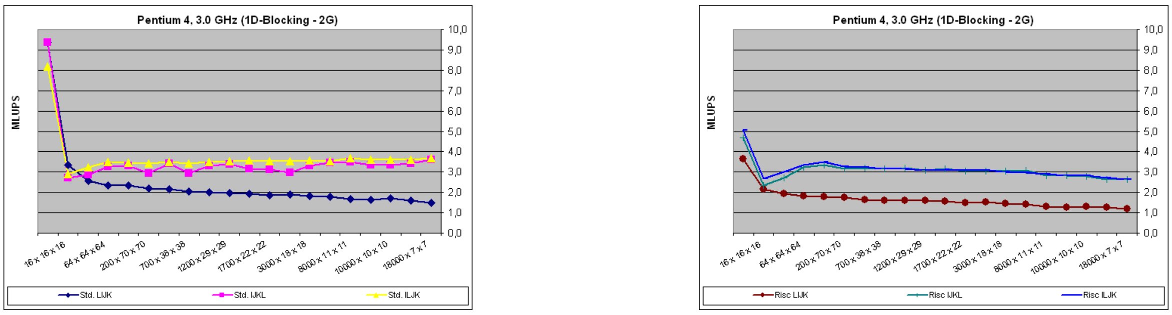


Figure A.15: 1D-Blocking, Two Grid layout on Test Platform 2 (P4, 3.0 GHz).

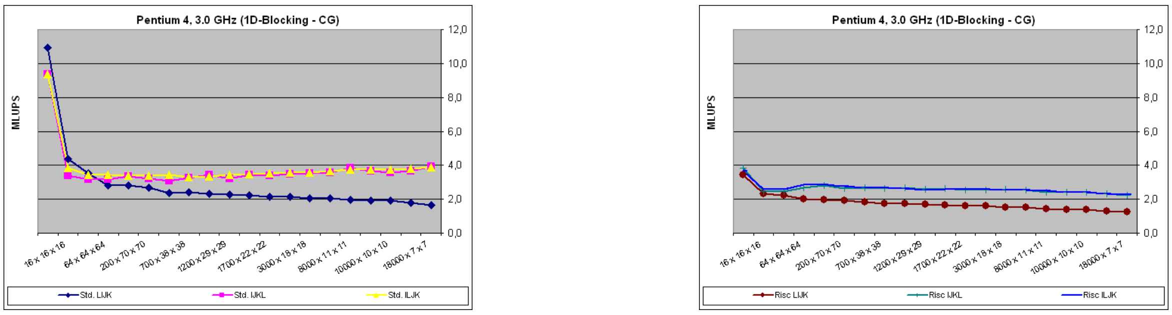
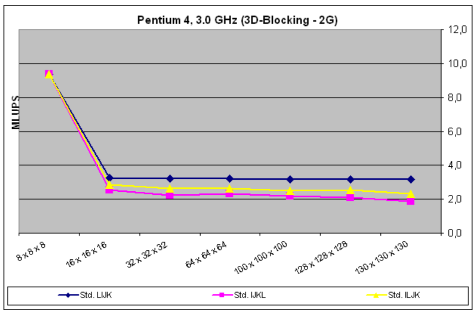


Figure A.16: 1D-Blocking, Compressed Grid layout on Test Platform 2 (P4, 3.0 GHz).



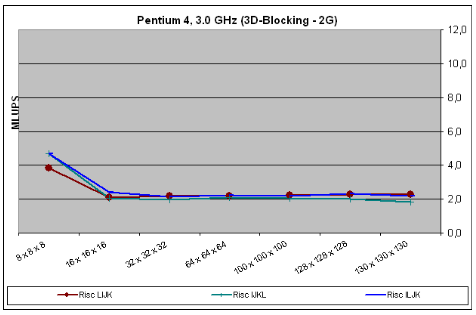
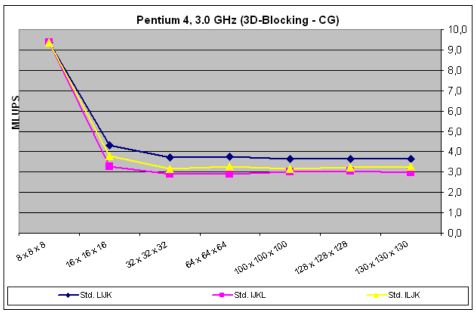


Figure A.17: 3D-Blocking, Two Grid layout on Test Platform 2 (P4, 3.0 GHz).



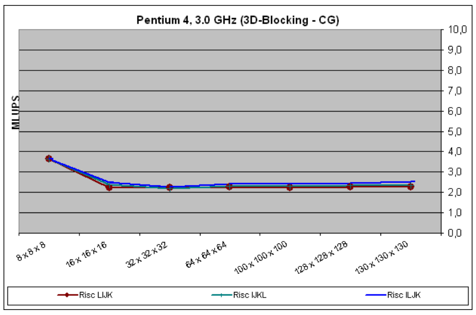
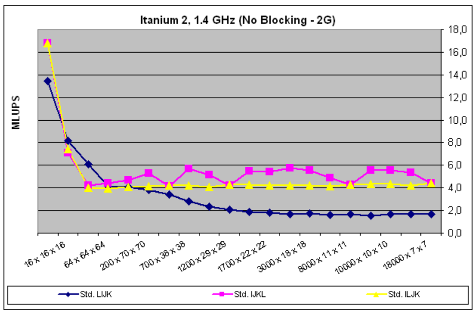


Figure A.18: 3D-Blocking, Compressed Grid layout on Test Platform 2 (P4, 3.0 GHz).

A.3 Test Platform 3 (Itanium 2, 1.4 GHz)



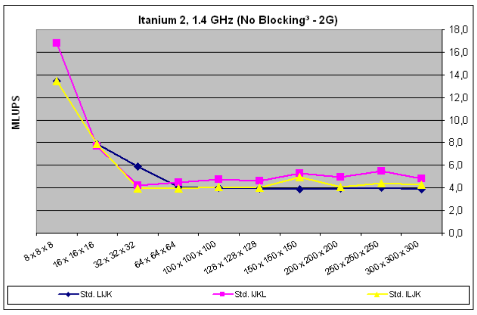
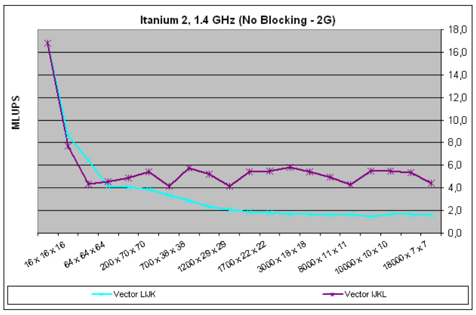


Figure A.19: Two Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz), Standard version.



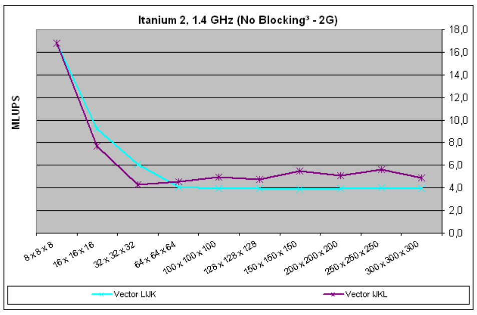
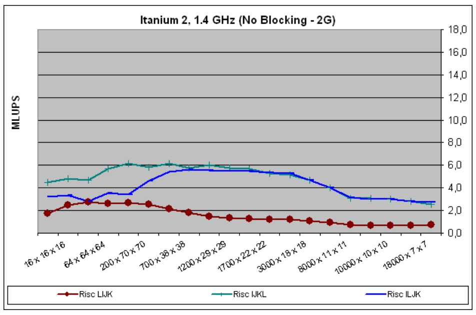


Figure A.20: Two Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz), Vector version.



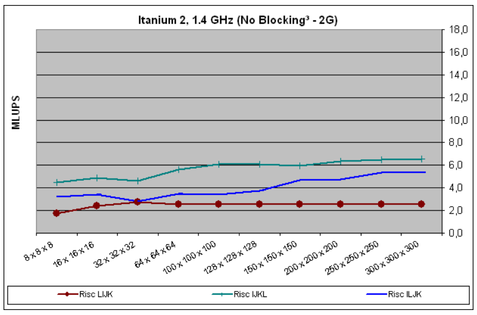


Figure A.21: Two Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz), RISC version.

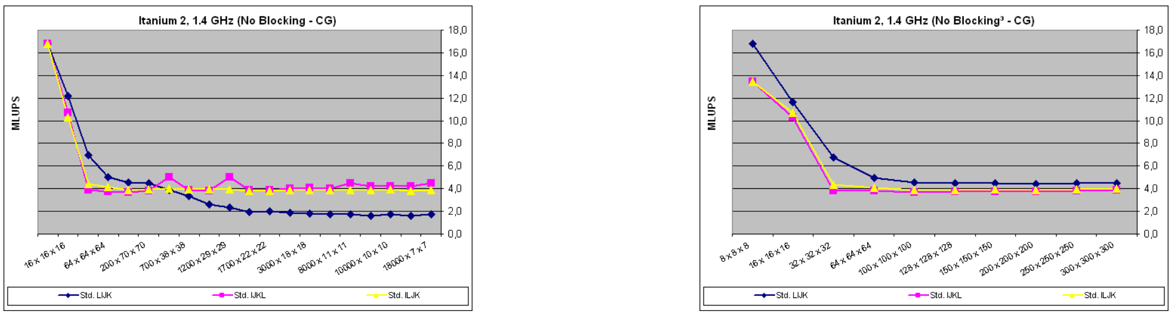


Figure A.22: Compressed Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz), Standard version.

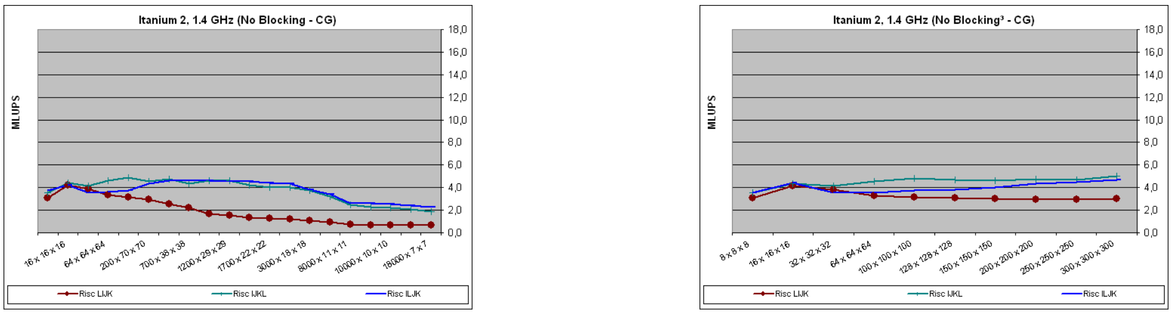


Figure A.23: Compressed Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz), RISC version.

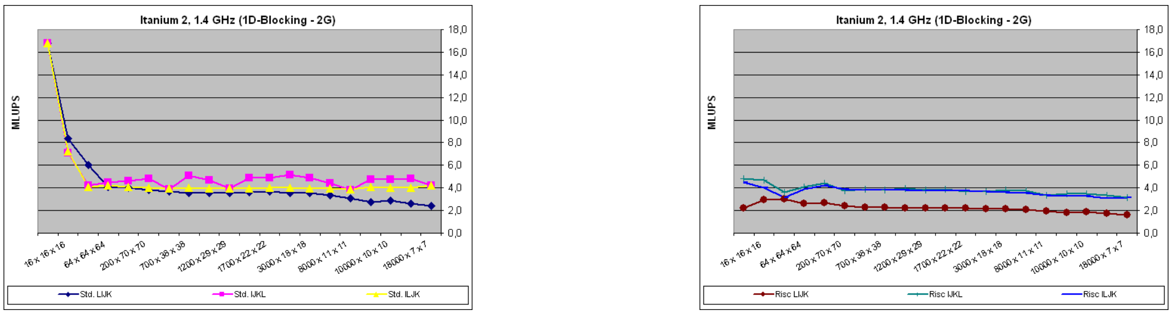


Figure A.24: 1D-Blocking, Two Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz).

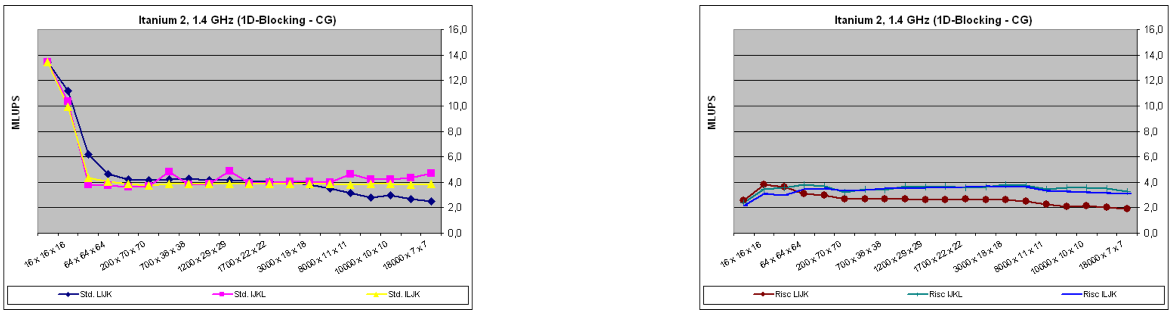
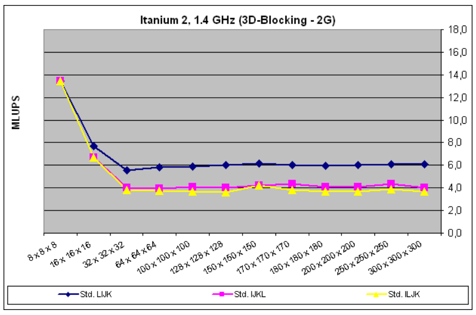


Figure A.25: 1D-Blocking, Compressed Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz).



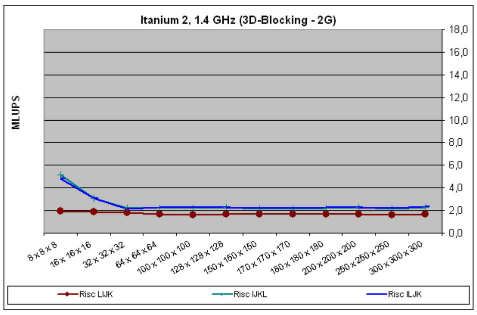
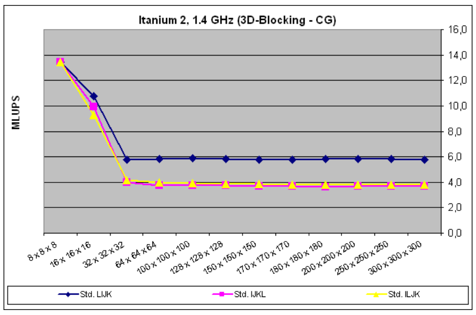


Figure A.26: 3D-Blocking, Two Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz).



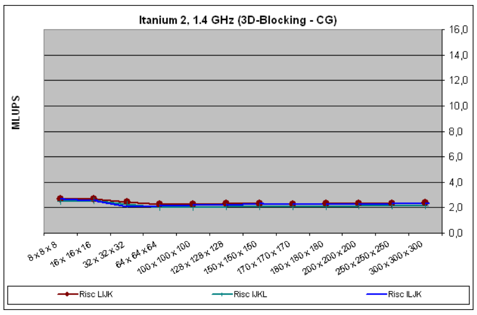


Figure A.27: 3D-Blocking, Compressed Grid layout on Test Platform 3 (Itanium 2, 1.4 GHz).

A.4 Test Platform 4 (Itanium 2, 1.3 GHz)

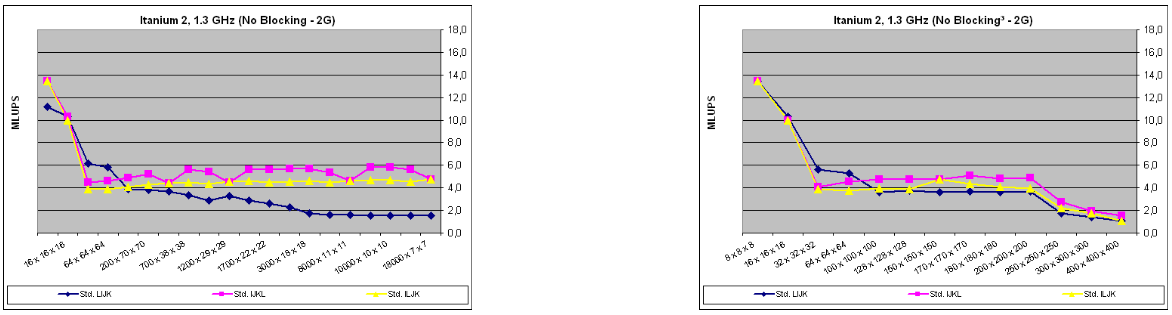


Figure A.28: Two Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz), Standard version.

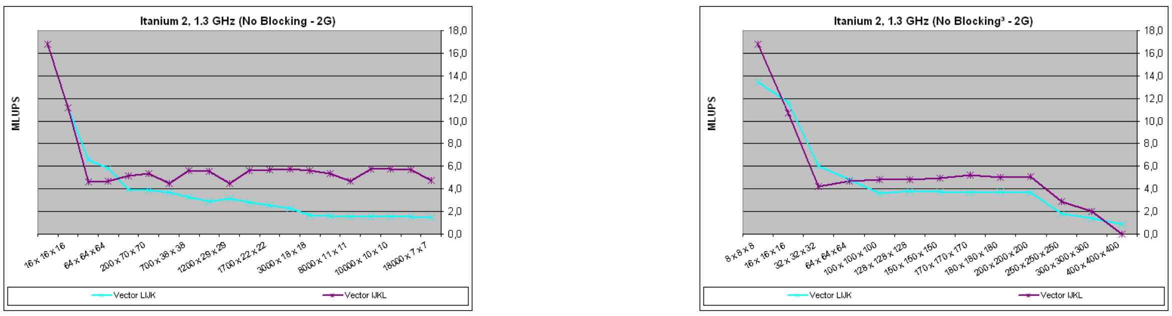


Figure A.29: Two Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz), Vector version.

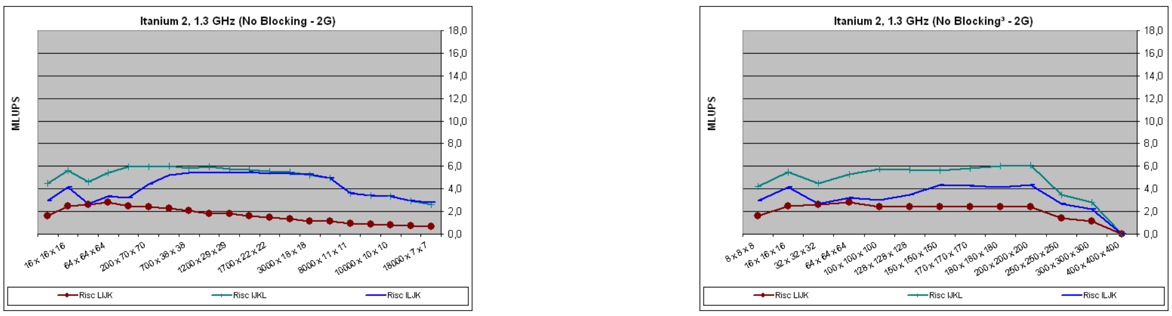


Figure A.30: Two Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz), RISC version.

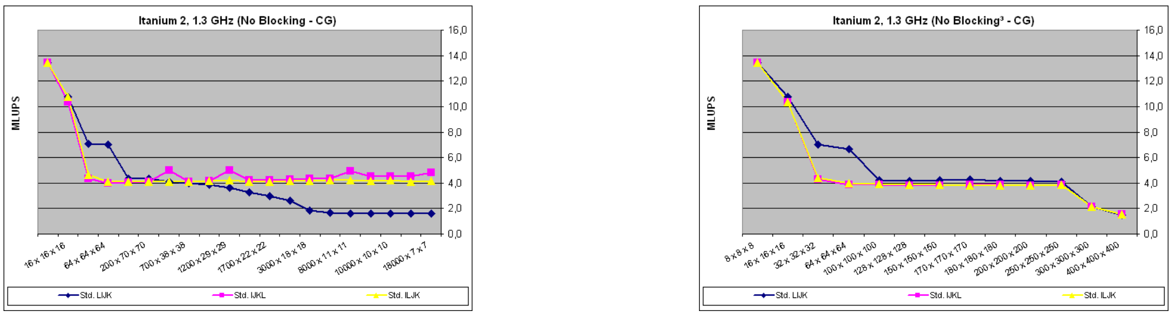


Figure A.31: Compressed Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz), Standard version.

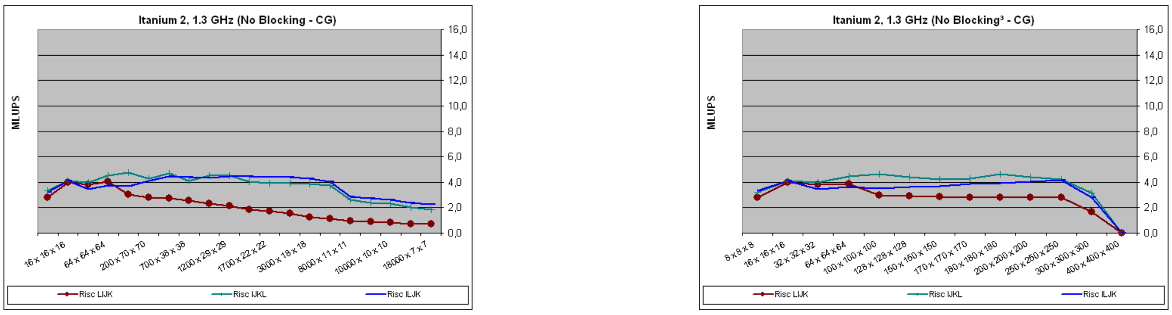


Figure A.32: Compressed Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz), RISC version.

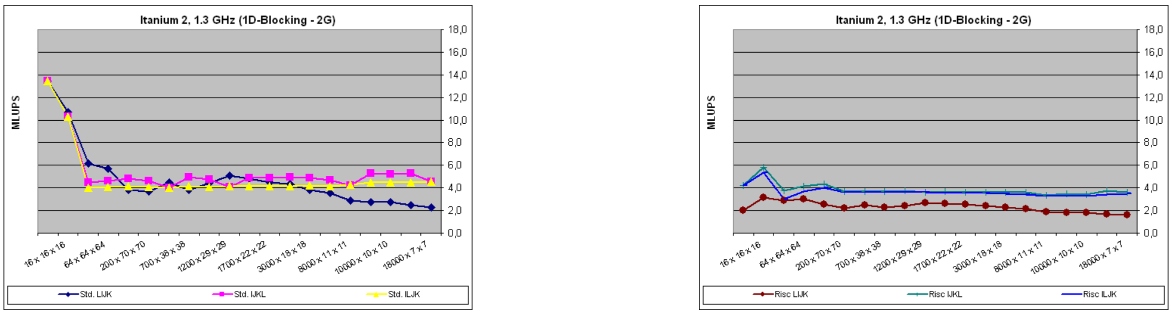


Figure A.33: 1D-Blocking, Two Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz).

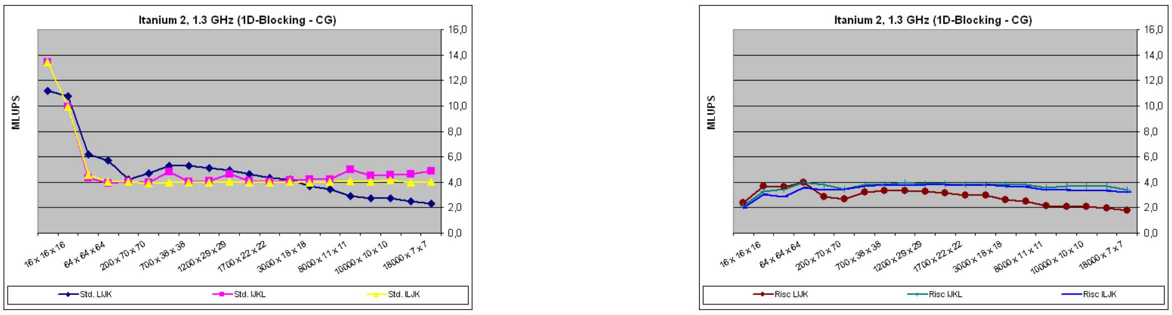
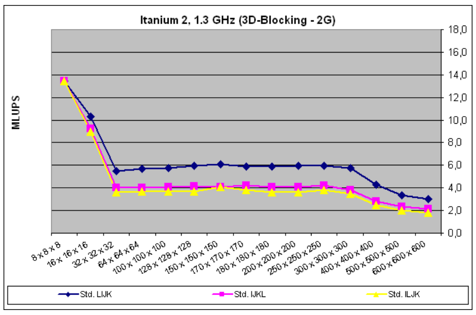


Figure A.34: 1D-Blocking, Compressed Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz).



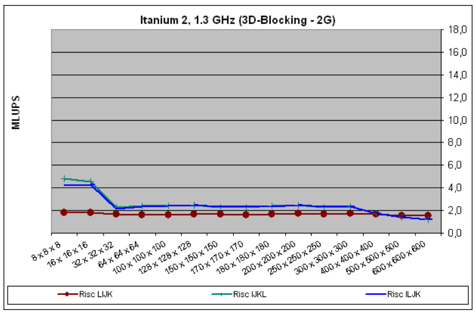
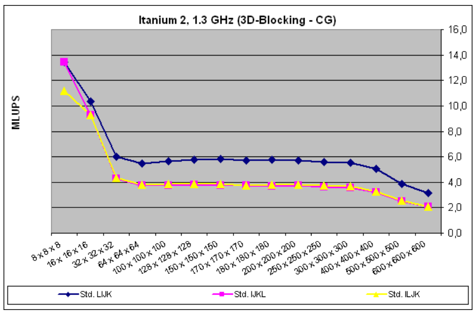


Figure A.35: 3D-Blocking, Two Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz).



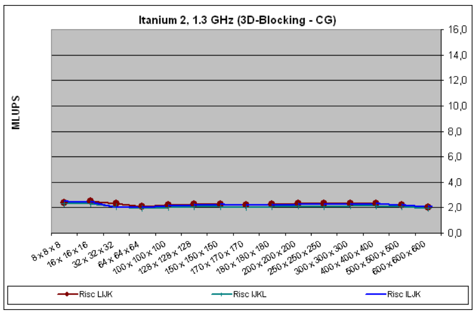


Figure A.36: 3D-Blocking, Compressed Grid layout on Test Platform 4 (Itanium 2, 1.3 GHz).

A.5 Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler

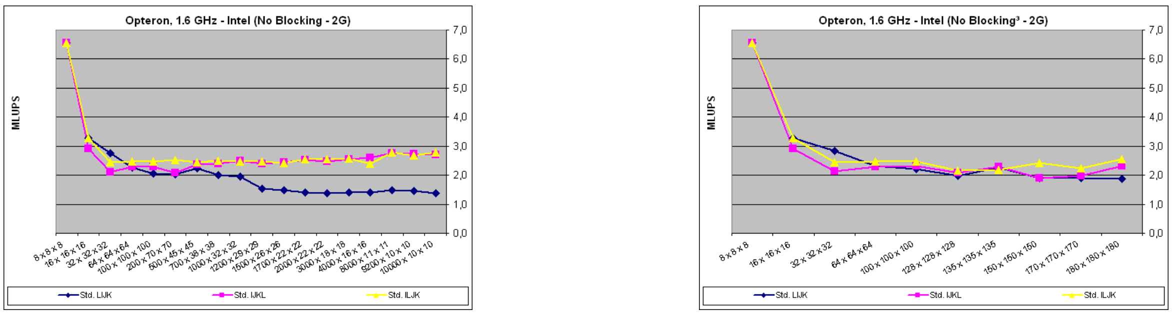


Figure A.37: Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler, Standard version.

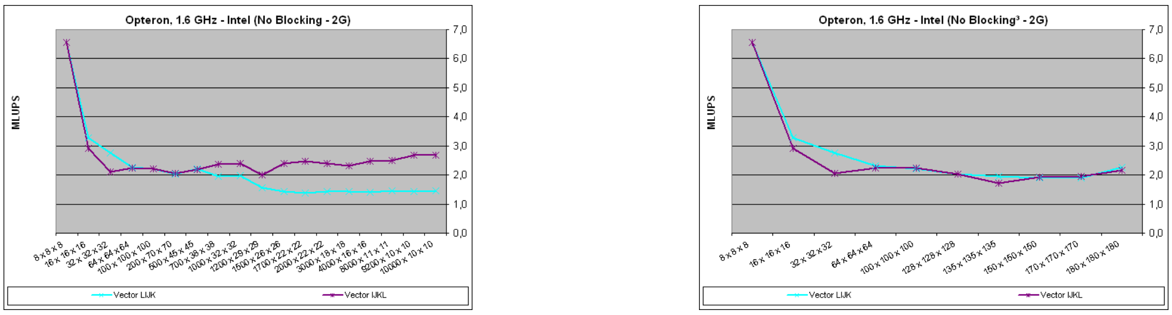


Figure A.38: Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler, Vector version.

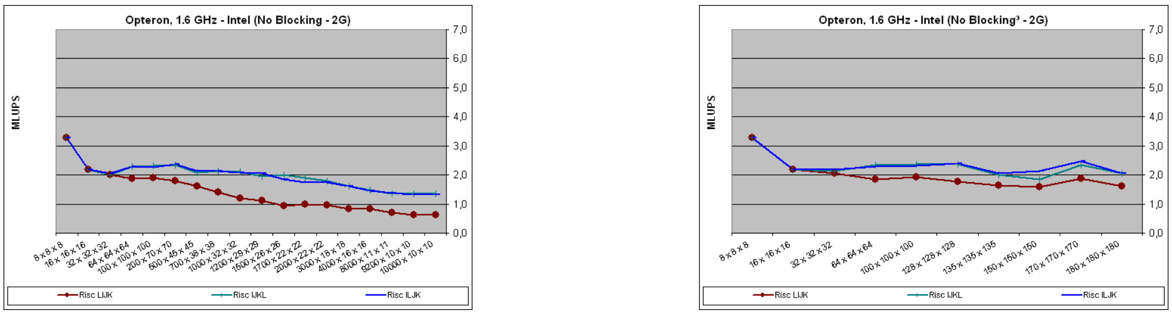


Figure A.39: Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler, RISC version.

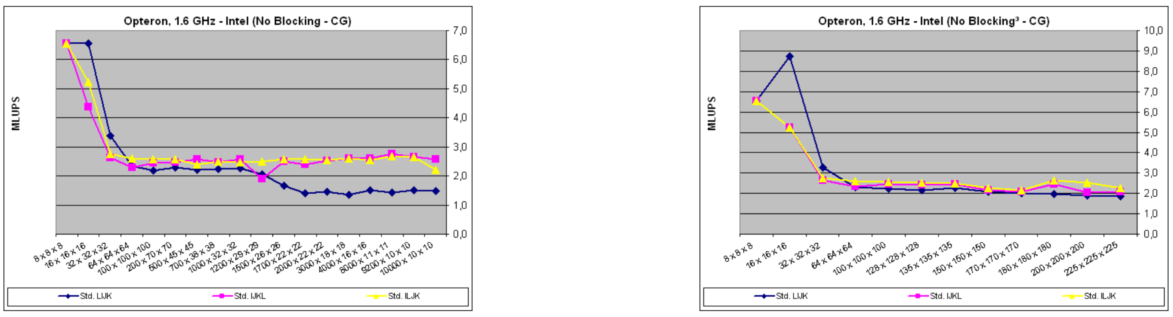


Figure A.40: Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler, Standard version.

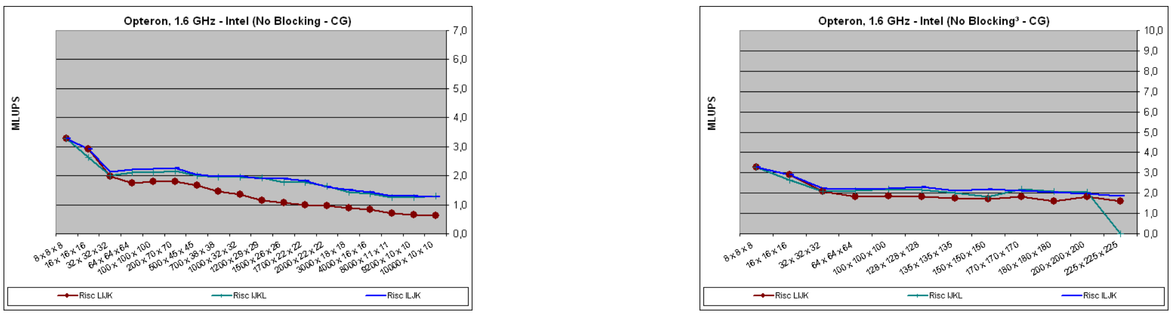


Figure A.41: Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler, RISC version.

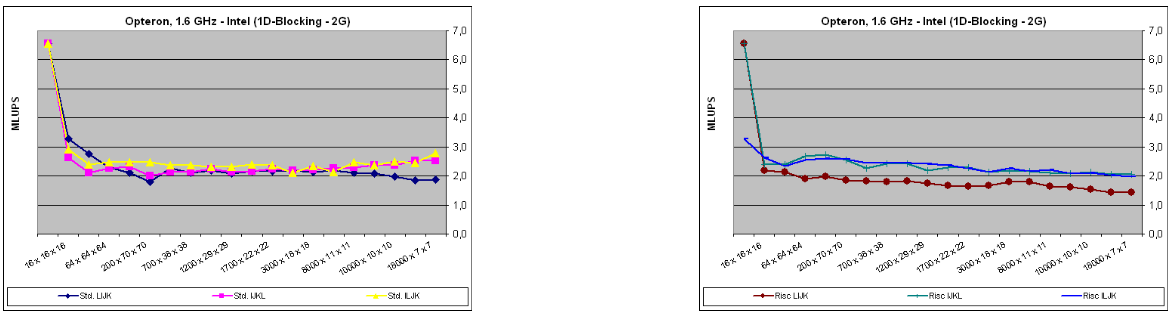


Figure A.42: 1D-Blocking, Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler.

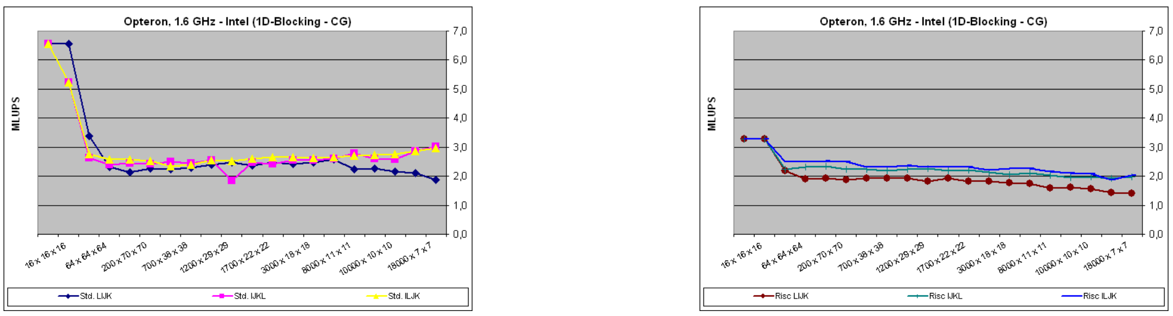


Figure A.43: 1D-Blocking, Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler.

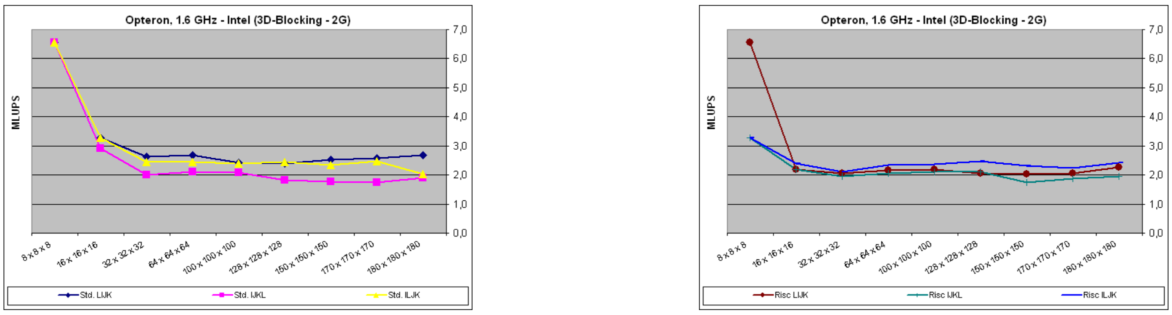


Figure A.44: 3D-Blocking, Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler.

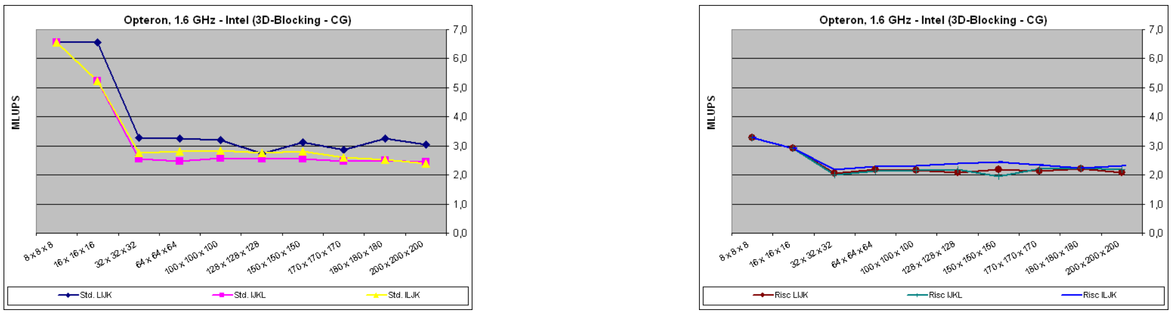
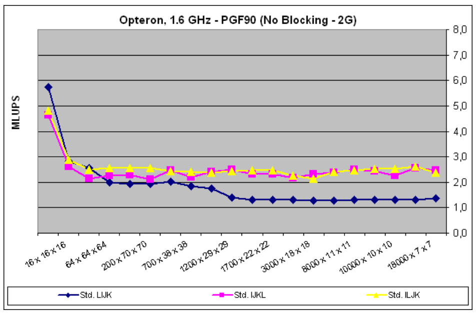


Figure A.45: 3D-Blocking, Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Intel Compiler.

A.6 Test Platform 5 (Opteron, 1.6 GHz), Portland Compiler



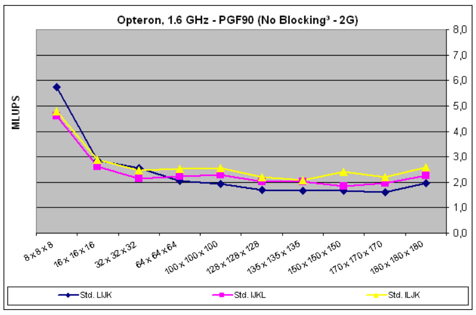
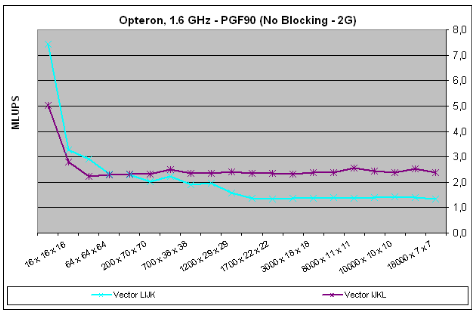


Figure A.46: Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler, Standard version.



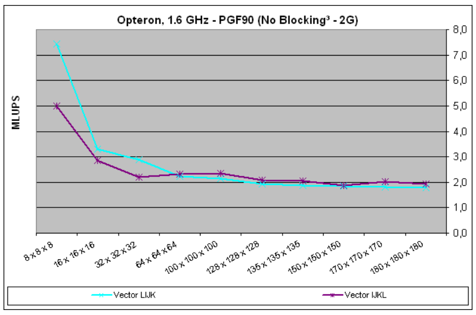
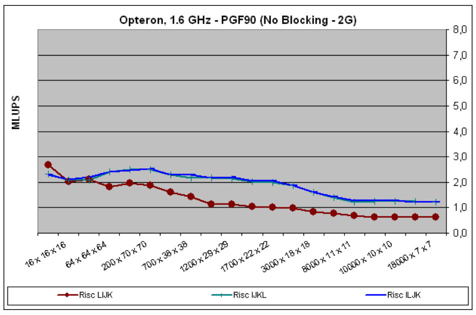


Figure A.47: Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler, Vector version.



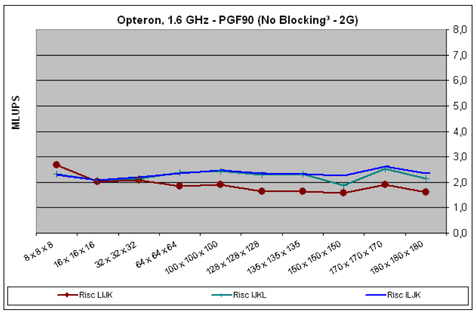
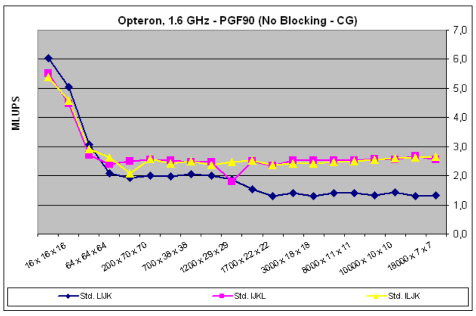


Figure A.48: Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler, RISC version.



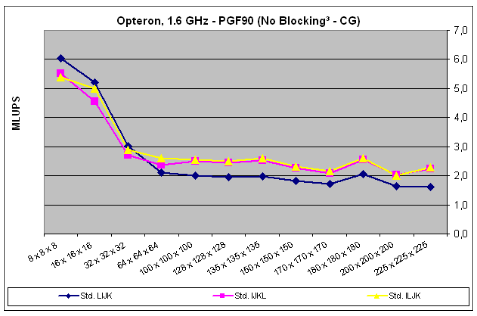
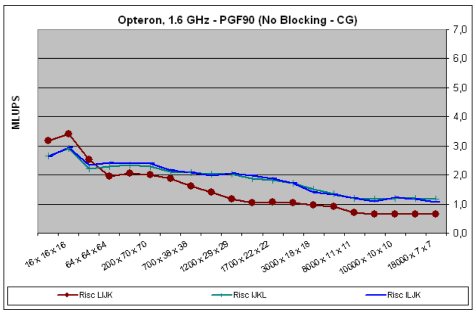


Figure A.49: Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler, Standard version.



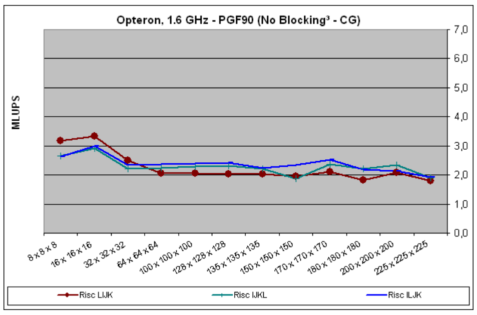
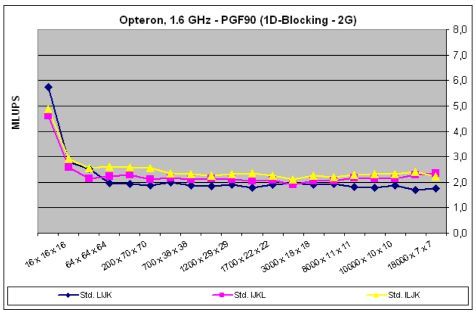


Figure A.50: Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler, RISC version.



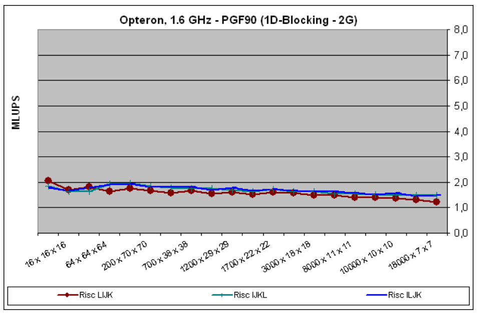
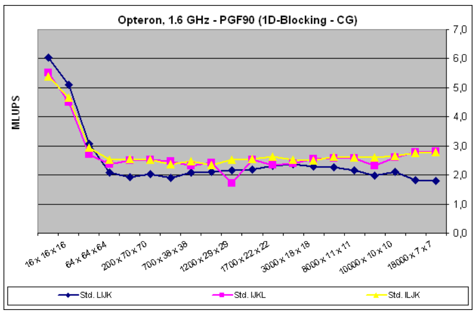


Figure A.51: 1D-Blocking, Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler.



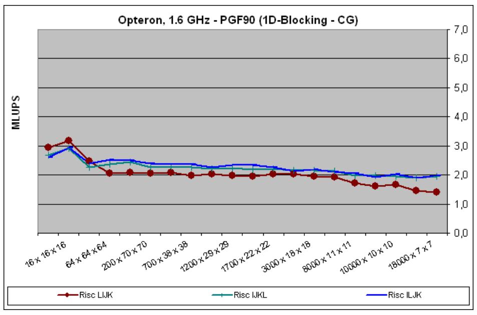
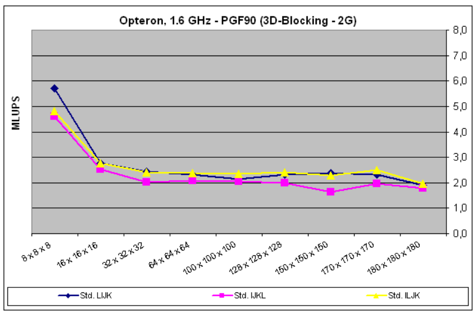


Figure A.52: 1D-Blocking, Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), PGI Compiler.



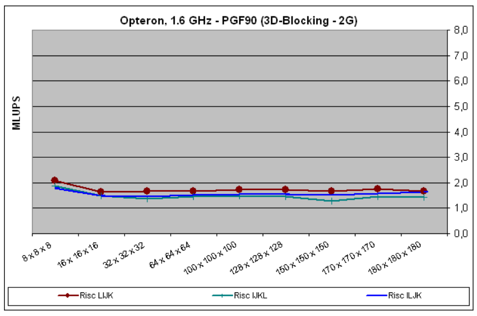
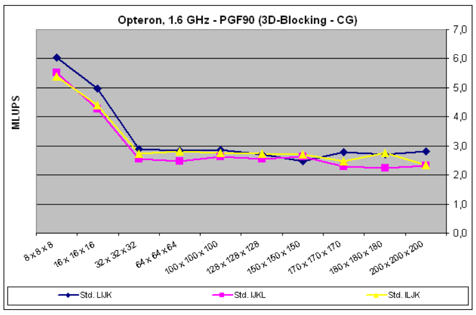


Figure A.53: 3D-Blocking, Two Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Portland Compiler.



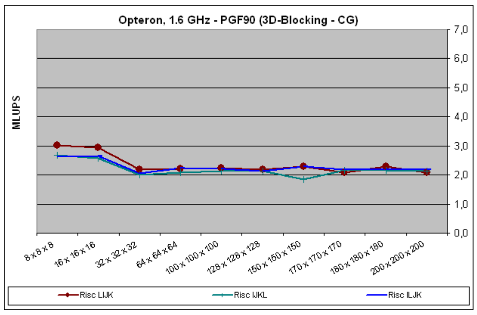


Figure A.54: 3D-Blocking, Compressed Grid layout on Test Platform 5 (Opteron, 1.6 GHz), Portland Compiler.

Appendix B

Description of Tools for Profiling and Cache Simulation

By profiling and simulation of important issues concerning code performance like cache misses and the number of floating point operations, causes for improvements or decreases of performance can be explained. For that, three different profiling tools and one cache simulating software were used, which are described shortly in the following.

B.1 Profiling

Profiling mostly was used to get hints for reasoning or developing better strategies. Following, a small introduction to the tools used is given. These are: *OProfile*, Intel's *VTune* and *lipfpm* of SGI's histx. Results of profiling and explanations how they influenced optimization work, are given in section 5.2.

B.1.1 Working with OProfile

OProfile, a free system profiler for Linux, is available on `oprofile.sourceforge.net`. It is a powerful tool which can read out all hardware counters that are provided by the processor. While an important drawback is, that *OProfile* can be executed under root account, only, it is very script friendly. So for this thesis, *OProfile* was used by using a perl script. Measuring with *OProfile* is fast, since it does not slow down the interesting executable, which is started by *OProfile*, very much. The results of measurements seem trustworthy, because they are stable from run to run. Generally, it is recommendable to run as few processes as possible on machine during the measurements, however, the intensity of influence on results by other running processes could not be determined. For this thesis, *OProfile* was used to confirm the results of *VTune*. Measurements were taken with version 0.7.1 on Pentium 4 platforms only.

B.1.2 Working with VTune

Intel's *VTune for Linux 2.0* is a very versatile tool. Besides sampling of hardware counters, it also provides a callgraph functionality, which enables tracking of subsequent function calls of a program. It does not need to be run under root account. This makes this tool more flexible than *OProfile*. For this thesis, only the sampling functionality was used. First, the user has to specify an activity, which defines the collectors to be counted, the executable that has to be started, and the duration time of one run. *VTune* starts the executable when user starts measuring, and ends it, when specified run time is exceeded. Many additional options are definable. For example, often it is advisable to set the size of sample buffer higher, and to disable the calibration mode and set sample sizes by oneself. By default, *VTune* runs the executable once for calibrating the sample size, and then starts measurement with the determined sample sizes. Though this is very comfortable, it takes often much time. If many collectors were specified, *VTune* determines by itself, which counters can be measured simultaneous, and which have to be measured in a separate run. In such cases, disabling self calibration can be advantageous. Often, experience suffices to specify sample sizes which produce results that are correct enough. Like other profiling tools, *VTune* measures the events system wide, too. Hence, it is recommendable to run as few processes as possible at same time. Nevertheless, it assigns the counted events to the different running processes reliably. The results are stored in a kind of database, where the entries are ordered by the number of run and grouped by the activities they belong to. While one can specify a name for each activity, unfortunately it is not possible to specify own names for each run. To view the sample results of a run, the *VTune* tool has to be used (it is not possible to parse the database directly). Although *VTune* supports two different ways to print out the data, none of both gives an easily parsable output. Writing scripts to automate measuring with *VTune* is relatively complicated. While the user friendliness is not very meritorious, the results are highly trustworthy. For this thesis, *VTune* was used for profiling on Pentium 4 architectures, only.

B.1.3 Working with lipfpm

lipfpm stands for Linux IPF Performance Monitor and is a tool that is integrated in histx packet, which is designed for SGI's Itanium systems. The used version 1.1 is installed on Test Platform 4 and, therefore, was used for profiling events on Itanium 2 processor. The main advantages of *lipfpm* are, that it is very user-friendly (it runs under every user account), fast and very handy, and that it produces output files that are easy to parse by scripts. In this output file *lipfpm* provides often additional informations to the requested counters. These are ratios like Instructions per Cycle or Requested Bandwidth by L3 Cache, and so on. For profiling with *lipfpm*, the executable must not be linked statically. Otherwise, profiling will not work. The only drawback of this tool is that it takes no care whether the combination of events to measure simultaneously is reasonable or not. For example the produced results when sampling L3 Misses and L2 Misses in same run, turned out to be not always trustable. For getting trustworthy results it is recommendable to measure each counter in separate runs. This is no real disadvantage because measurements are done very fast. This tool is very easy to embed in scripts for automation. Because of its user friendliness, *lipfpm* was the preferred tool for profiling.

B.2 Simulation with kcachegrind

For getting better insight in strengths and weaknesses of the algorithm, it is useful to simulate the performance limiting events, instead of measuring. For this, *kcachegrind* ([Wei03]) is a practical tool that is able to simulate read and write misses for L1 and L2 cache, fetched instructions, reads and writes. For this thesis, simulation by *kcachegrind* was used with same intention like profiling. The results should be explained and new strategies developed. *kcachegrind* simulates a cache architecture with separated L1 caches for instructions and data and combined L2 cache, which size, cache line length and associativity is equal to that of the Pentium 4 architecture used [Wei04]. Since the simulated caches are write-through with write-allocate, the simulation performs exactly the same for read and write accesses, and thus, it is not comparable absolutely with Pentium 4's cache structure. While some results could be included in research for this thesis, results concerning L2 misses cannot taken seriously.

kcachegrind is split into two parts: One that collects all information by running the executable, and one that visualizes the data. For collecting the values, the executable has to be compiled with

`-g` option. The subprogram *calltree* runs and examines the executable, that is specified by parameter. It is recommended to use the additional flags "`--trace-jump=yes`" and "`--dump-instr=yes`". The analysis of executable takes much longer than a normal run. At the end, an output file named "cachegrind.out.*pid*" is created. This file has to be opened with *kcachegrind* for visualization. For read and write misses on L1 and L2 caches, fetched instructions and a cost function that includes all events, a summary is given, which is subdivided into its parts assigned to the called subroutines of executable. In a comfortable GUI the single assembly instructions are listed, including the events that occurred in the corresponding line. If the source codes of executable are available, *kcachegrind* assigns the events even to the instructions of high level language. Furthermore graphical trees of callers and callees can be examined.

Appendix C

Memory Visualizer MemPHis

Very often it is hard to imagine the complex activities related to cache and memory interaction by only examining the code and interpreting profiling results. Thus, the search for a possibility to visualize memory accesses resulted in developing a memory visualizer. The idea is, that each read or write access to memory should be made visible in a manner that is easier to interprete. Thereby no attention to influence of cache architecture is taken into account. Visualizing the location and stride of memory accesses could help in estimating cache behavior, understanding performance results and evolving new ideas.

C.1 Memory Visualizer as a Library

Since such a visualizing tool should be very handy and not too complex, it was decided to create a library. In contrast to a program, that parses the source code of a program, and, thus, is only capable to interpret one specific programming language, a library can be built-in in any program. Moreover, including the functionality by subroutine calls gives the user most freedom in usage. Hence, the *MemPHis* library was developed. It is written in C to be compatible to most other languages. It records the position of memory accesses on the interesting array in a colored bitmap. The user has to add only few additional commands to his code for the allocation and initialization of a buffer, used by the *MemPHis* methods. Since the interesting array can consist of any number of dimensions, at initialization of the buffer the boundaries have to be known. For each read or write access on the interesting array, a subroutine of the *MemPHis* library is called instead. This replacement does not influence the correctness of the user's code, since the subroutines return the read value as if it was read directly from the array. In an ideal case, the user has to replace simply each read access on the array by the subroutine call using textual search and replace functions. At this time, only `double` and `int` data types are supported for interesting array. To visualize the chronology of accesses, the user can determine which color is used in bitmap for each memory access. By calling other subroutines, the user can initiate a dump of the current image to a file or clear the whole image to white. The output files are in 24 bit BMP format with line padding. For easier interpretation, the width of the image corresponds to the length of a cache line, by default. This is defined in initialization and can be adapted as well by the user.

C.2 How to Use MemPHis

This section gives an introduction of how to use the *MemPHis* library. For better understanding it might be useful to have a look at the source code of *MemPHis* library in section D.2 or compare with implementation example in *LBMKernel* in section C.4.

Since the library needs own memory space, the calling program has to allocate enough memory first. To find out, how much memory has to be allocated, the function `mv_getsize()` must be used. This function receives the size of interesting array (in Bytes), the size of used data type in array and the size of the boundary description arrays. For initialization of buffer, the *MemPHis* library needs a full description of number and size of dimensions of the array. For this, the user has to create two arrays, where the first one contains the lower bounds of the dimensions and the second the upper bounds, respectively. For termination, last entry of these arrays has to contain the value `-1`. First entry of these description arrays must describe closest dimension of array, that lies consecutively in memory with unit stride (in Fortran: first dimension of array, in C: last dimension of array). After having received the needed size for buffer, the calling program has to allocate the memory. By calling `mv_init()` the buffer has to be initialized. `mv_init()` receives pointers to the buffer, its size, the size of the array and the description arrays. Additionally it has to know the data type of array and the length of cache line (in Bytes). Thus, the width of image in pixels will be `cachelinesize / datatype`.

After initialization phase is finished, every interesting memory access on array has to be replaced by a function call of `mv_step_db()` for a `double` array, or `mv_step_int()` for an `int` array, respectively. These functions receive as parameters a pointer to the buffer and to the array, an `int` value which describes the marking color in bitmap, and the indexes that specify which location of array is accessed. The returned value is the `double` (or `int`) stored on specified location in array. The `color` is an integer value, that describes RGB value according to the HTML standard, where first byte is uncared, second stands for red, third for green and last for blue, respectively.

Whenever the user wishes, the actual state of buffer can be dumped in a bitmap file by calling `mv_flush()`. For resetting the image to white, the function `mv_reset()` can be used.

C.3 Details to Source of MemPHis

In Algorithm D.2 (Appendix D) lines 6 to 26 describe the internal structure of the buffer. The header, whose constant part consists of 20 Bytes, contains information about the total length of buffer, the length of header (which is the offset to access the body of the buffer), data type and dimensions of user's array (dimensions are stored without terminating `-1` values of description arrays!). Last two bytes of header are empty at this time, to enable any format extensions for eventually following versions. At location `StartAddressOfBuffer+LengthOfHeader` the body of buffer begins. Here the colors for each pixel of image are stored in ascending order according to the array description, using three bytes per pixel (for red, green and blue). The constant positions of meta data in buffer are named by preprocessor macros (lines 28 to 34).

The internal functions `storeIntToBuffer` and `loadIntFromBuffer` are called by the *MemPHis* library functions to access the right position of buffer. Function `mv_getsize()` calculates the needed size of buffer by computing the size of body and adding the constant size of header to its variable one, basing on the informations received by parameters.

The initialization of buffer is done by `mv_init()`. This function first calculates the size of header basing on received informations, and then checks if allocated buffer has the correct size. If requested data type of array is supported, the header of buffer will be filled with information and image data in the body will be initialized to white.

The `mv_step` functions (in Algorithm D.2 only version for `double`, `mv_step_db()`, is printed) can receive a variable number of parameters for being able to process the access on a variably dimensional array. First, they determine the memory location (which corresponds to location in body of buffer) basing on the passed indexes and the length of dimensions of user's array (stored in header of buffer). After extracting red, green and blue value out of parameter color, they are stored to the corresponding location in buffer. The function returns the element of user's array that resides on the position described by index parameters. Function `mv_step_int()` contains exactly same code as function `mv_step_db()` excepting the declaration for return type.

Function `mv_reset()` reinitializes the body of buffer by setting all color information back to white.

Dumping the body data of buffer to a bitmap file is done by `mv_flush()`. First, a file name is generated out of current operating system time stamp. Then this function writes the standard header for BMP images into the file. Therefore, as no compression is used, the type of BMP format is set to 21 bit, and horizontal and vertical resolution is set to 2835 pixels per inch. For the size of file first is written zero. It is inserted in the end of function. As no documentation gave a valid description of BMP format, by experiments a kind of zero padding turned out to be useful, so that the file is accepted as valid BMP by most graphic viewers: Instead of storing each pixel four byte wise, only three bytes per pixel are used. The end of each image line will then be padded with zeros for having a multiple of four for the number of bytes per line (Width of one image line is a natural number of integers). After last line of picture is filled with white pixels, the correct file size is inserted, and the file will be closed. Note that the lines in image are descending, such that address 0 of array corresponds to the lowest left pixel of picture.

For being compatible at least with calling programs written in Fortran and C, all functions receive their data by pointers. Fortran compilers implement by default call by reference, so this makes implementation in Fortran programs easier. For avoiding problems when linking *MemPHis* library into Fortran programs, the names of all public functions end with an underbar ('_').

C.4 Implementation of MemPHis in LBMKernel

As described in section C.2, the implementation of *MemPHis* library is relatively easy. In *LBMKernel*, the *MemPHis* functionality is included when Makefile specifies -DMEMPHIS flag and links the object file of *MemPHis* library. For clearness, in the following example only implementation of *MemPHis* for Two Grid layout with LIJK data layout is shown. First, some additional declarations are needed. Algorithm C.1 shows that the description arrays `Begins` and `Ends` are declared, `theBuffer` as buffer for *MemPHis*, and the variables `buffersize` and `color`. In addition, the external subroutines of *MemPHis* library are declared. After that, the description arrays receive the boundary information of `pdf` array. First the 19 directions (0..18), then the three dimensions and last the two grids. Both arrays are terminated with an element containing the value `-1`.

Next is to request the size for buffer, allocation and initialization. This is shown in Algorithm C.2: Calling the function `mv_getsize()` with information about size of `pdf` array, its data type and the description arrays, the needed size for the buffer is returned. Next step is, to allocate `theBuffer` with needed number of Bytes. The buffer will be initialized by calling `mv_init()`. It receives the buffer, its size, the size of `pdf` array and the description arrays `Begins` and `Ends` containing the information about boundaries and dimensions of `pdf` array. The size of data type (`double`) is eight Bytes, the

Algorithm C.1: MemPHis in LBMKernel: Declarations and description arrays

```
#ifdef MEMPHIS
  !MemPHis - Declarations

  logical, dimension(:), allocatable :: theBuffer
  integer (I4B) :: buffersize
  integer (I4B), dimension(0:5) :: Begins
  integer (I4B), dimension(0:5) :: Ends
  integer (I4B) :: color, color2

  integer, external :: mv_getsize
  integer, external :: mv_init
  real, external :: mv_step_db

  Begins(0)=0
  Begins(1)=0
  Begins(2)=0
  Begins(3)=0
  Begins(4)=0
  Begins(5)=-1
  Ends(0)=18
  Ends(1)=iEnd+1
  Ends(2)=jEnd+1
  Ends(3)=kEnd+1
  Ends(4)=1
  Ends(5)=-1
#endif
```

Algorithm C.2: MemPHis in LBMKernel: Allocation and Initialization of buffer

```
#ifdef MEMPHIS
  buffersize=mv_getsize(sizeof(pdf),8,sizeof(Begins));
  allocate(theBuffer(buffersize))

  i=mv_init(theBuffer, buffersize, sizeof(pdf),128,8,Begins,Ends)
  if (i.ne.0) then
     write (*,*) 'Fehler bei mv_init: ',i
     return
  end if

  color=13404330
  color2=1179443
#endif
```

cache line size is chosen to 128 Bytes. If `mv_init()` returns a value different from zero, the return value can give information about the reason of failure. The two colors used in this example are purple for reading values and green for written values.

Algorithm C.3 shows examples how to use function `mv_step_db()`. The read access on `pdf` array is simply replaced by the function call. The function needs additional arguments like `theBuffer`, the `pdf` array, and the color for this pixel. Unfortunately, the write accesses on `pdf` array cannot be replaced by a function call, if correctness of code shall be assured. Thus, the `mv_step_db` is called separately, here.

At the end of collision subroutine, the image data of the current time step is dumped to a file by calling `mv_flush()` (see Algorithm C.4). Since the calling program allocated the buffer, it is important that it deallocates it, too.

Algorithm C.3: MemPHis in LBMKernel: Accessing the array

```
  d1     = mv_step_db(theBuffer,pdf,color,IJKL_R(i,j,k, 1,tNow)) + &
           mv_step_db(theBuffer,pdf,color,IJKL_R(i,j,k, 7,tNow)) + &
           mv_step_db(theBuffer,pdf,color,IJKL_R(i,j,k, 8,tNow))
  d2     = mv_step_db(theBuffer,pdf,color,IJKL_R(i,j,k, 3,tNow)) + &
           mv_step_db(theBuffer,pdf,color,IJKL_R(i,j,k, 4,tNow)) + &
           mv_step_db(theBuffer,pdf,color,IJKL_R(i,j,k, 5,tNow))

  ...

  pdf(IJKL_W(i  ,j  ,k ,0,tNext)) = pdf(IJKL_R(i,j,k, 0,tNow)) * ImOmega + ne0
  call mv_step_db(theBuffer,pdf,color2,IJKL_W(i  ,j  ,k , 0,tNext))

  pdf(IJKL_W(i+1,j+1,k ,1,tNext)) = pdf(IJKL_R(i,j,k, 1,tNow)) * ImOmega + ne1
  call mv_step_db(theBuffer,pdf,color2,IJKL_W(i+1,j+1,k , 1,tNext))

  pdf(IJKL_W(i  ,j+1,k ,2,tNext)) = pdf(IJKL_R(i,j,k, 2,tNow)) * ImOmega + ne2
  call mv_step_db(theBuffer,pdf,color2,IJKL_W(i  ,j+1,k , 2,tNext))

  pdf(IJKL_W(i-1,j+1,k ,3,tNext)) = pdf(IJKL_R(i,j,k, 3,tNow)) * ImOmega + ne3
  call mv_step_db(theBuffer,pdf,color2,IJKL_W(i-1,j+1,k , 3,tNext))

  pdf(IJKL_W(i-1,j  ,k ,4,tNext)) = pdf(IJKL_R(i,j,k, 4,tNow)) * ImOmega + ne4
  call mv_step_db(theBuffer,pdf,color2,IJKL_W(i-1,j  ,k , 4,tNext))

  pdf(IJKL_W(i-1,j-1,k ,5,tNext)) = pdf(IJKL_R(i,j,k, 5,tNow)) * ImOmega + ne5
  call mv_step_db(theBuffer,pdf,color2,IJKL_W(i-1,j-1,k,5,tNext))

  ...
```

Algorithm C.4: MemPHis in LBMKernel: Dump data to Bitmap file and deallocate buffer

```
#ifdef MEMPHIS
  call mv_flush(theBuffer)
  deallocate(theBuffer)
#endif
```

Appendix D

Algorithms

D.1 Standard LBM Collision Step

The following algorithm shows the standard implementation with non-cache optimizations already implemented: Arithmetic optimization by precalculating several values and reordering of operations to give the compiler hints for scheduling.
This code shows no implementation of blocking, Lid Driven Cavity, *MemPHis* or Compressed Grid. By defining labels IJKL_LAYOUT and LIJK_LAYOUT in right manner, the Makefile produces the three subroutines

- col_standard_LIJK_2G,
- col_standard_IJKL_2G and
- col_standard_ILJK_2G,

where the abbreviation '2G' denotes the Two Grid layout.

Algorithm D.1: Collide step of LBM (Standard-Version)

```
#ifdef IJKL_LAYOUT
subroutine col_standard_ijkl_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField ,rho,acc)
#define IJKL_R(i,j,k,l,tN) i,j,k,l,tN
#define IJKL_W(i,j,k,l,tN) i,j,k,l,tN
#else
#ifdef ILJK_LAYOUT
subroutine col_standard_iljk_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField ,rho,acc)
#define IJKL_R(i,j,k,l,tN) i,l,j,k,tN
#define IJKL_W(i,j,k,l,tN) i,l,j,k,tN
#else
subroutine col_standard_lijk_2G(pdf,tNow,tNext,omega,iEnd,jEnd,kEnd,obstacleField ,rho,acc)
#define IJKL_R(i,j,k,l,tN) l,i,j,k,tN
#define IJKL_W(i,j,k,l,tN) l,i,j,k,tN
#endif
#endif

  implicit none

  integer, parameter :: DP  = kind(1.0D0)
  integer, parameter :: I4B = selected_int_kind(9)

  integer (I4B) :: tNow,tNext
  integer (I4B) :: iEnd,jEnd,kEnd
  integer (I4B) :: i,j,k

  real (dp), parameter :: frac1_3      =   1.0_dp /  3.0_dp
  real (dp), parameter :: frac2_3      =   2.0_dp /  3.0_dp
  real (dp), parameter :: frac1_8      =   1.0_dp /  8.0_dp
  real (dp), parameter :: frac1_18     =   1.0_dp / 18.0_dp
  real (dp), parameter :: frac1_36     =   1.0_dp / 36.0_dp
  real (dp), parameter :: neg_frac4_3  = - 4.0_dp /  3.0_dp

  real (dp) :: omega
  real (dp) :: ux, uy, uz, d_tmp, id_tmp
  real (dp) :: omega_2, ImOmega
  real (dp) :: d1, d2, d3, d4
  real (dp) :: usq, usqn, usqn1, usqn3
  real (dp) :: coeff_1, coeff_2, coeff_3, coeff_4
  real (dp) :: ui1, ui2, ui3, ui4, ui5, ui6, ui7, ui8
  real (dp) :: ne0, ne1, ne2, ne3, ne4, ne5, ne6, ne7, ne8
  real (dp) :: ui9, ui10, ui11, ui12, ui13
  real (dp) :: ne9, ne10, ne11, ne12, ne13, ne14, ne15, ne16, ne17, ne18
  real (dp) :: acc
  real (dp) :: rho

  logical,   dimension(0:iEnd+1,0:jEnd+1,0:kEnd+1) :: obstacleField

  real (dp), dimension(IJKL_R(0:iEnd+1,0:jEnd+1,0:kEnd+1,0:18,0:1)) :: pdf


  omega_2 = 2.0_dp * omega
  ImOmega = 1.0_dp - omega

! update whole Grid one time
  do k = 1, kEnd, 1
    do j = 1, jEnd, 1
      do i = 1, iEnd, 1
        If .not. obstacleField(i,j,k) then
          ! Read particle velocity distribution from current cell
          d1    = pdf(IJKL_R(i,j,k, 1,tNow)) + pdf(IJKL_R(i,j,k, 7,tNow)) + &
                  pdf(IJKL_R(i,j,k, 8,tNow))
          d2    = pdf(IJKL_R(i,j,k, 3,tNow)) + pdf(IJKL_R(i,j,k, 4,tNow)) + &
                  pdf(IJKL_R(i,j,k, 5,tNow))
          d3    = ...
          d4    = ...
          d_tmp = pdf(IJKL_R(i,j,k, 0,tNow)) + pdf(IJKL_R(i,j,k, 2,tNow)) + &
                  pdf(IJKL_R(i,j,k, 6,tNow)) + d1 + d2 + d3 + d4
          id_tmp = 1.0_dp / d_tmp

          ! Compute velocities
          ux = d1 - d2 + &
               pdf(IJKL_R(i,j,k,10,tNow)) + pdf(IJKL_R(i,j,k,15,tNow)) - &
               pdf(IJKL_R(i,j,k,12,tNow)) - pdf(IJKL_R(i,j,k,17,tNow))
          uy = ...
          uz = ...
          ux = ux * id_tmp
          uy = uy * id_tmp
          uz = uz * id_tmp

          coeff_1 = d_tmp * frac1_8 * omega
          coeff_2 = ...
          ...

          usq = ( ux * ux + uy * uy + uz * uz )
          usqn = ...
          usqn1 = usqn * omega
          usqn3 = usqn * omega_2

          ui1  = ux + uy
          ui2  = ...
          ...

          ! Compute non-equilibrium distribution
          ne0  = d_tmp * (frac1_3 - 0.5_dp * usq) * omega
          ne1  = coeff_1 * ui1 * (frac2_3 + ui1 ) + usqn1
          ne2  = ...
          ...
          ne17 = ...
          ne18 = ne11 + coeff_2 * ui11

          ! Push new values in adjacent cells
          pdf(IJKL_W(i  ,j  ,k  , 0,tNext)) = pdf(IJKL_R(i,j,k, 0,tNow)) * ImOmega + ne0
          pdf(IJKL_W(i+1,j+1,k  , 1,tNext)) = pdf(IJKL_R(i,j,k, 1,tNow)) * ImOmega + ne1
          pdf(IJKL_W(i  ,j+1,k  , 2,tNext)) = pdf(IJKL_R(i,j,k, 2,tNow)) * ImOmega + ne2
          pdf(IJKL_W(i-1,j+1,k  , 3,tNext)) = pdf(IJKL_R(i,j,k, 3,tNow)) * ImOmega + ne3
          pdf(IJKL_W(i-1,j  ,k  , 4,tNext)) = pdf(IJKL_R(i,j,k, 4,tNow)) * ImOmega + ne4
          pdf(IJKL_W(i-1,j-1,k  , 5,tNext)) = pdf(IJKL_R(i,j,k, 5,tNow)) * ImOmega + ne5
          pdf(IJKL_W(i  ,j-1,k  , 6,tNext)) = pdf(IJKL_R(i,j,k, 6,tNow)) * ImOmega + ne6
          pdf(IJKL_W(i+1,j-1,k  , 7,tNext)) = pdf(IJKL_R(i,j,k, 7,tNow)) * ImOmega + ne7
          pdf(IJKL_W(i+1,j  ,k  , 8,tNext)) = pdf(IJKL_R(i,j,k, 8,tNow)) * ImOmega + ne8
          pdf(IJKL_W(i  ,j  ,k+1, 9,tNext)) = pdf(IJKL_R(i,j,k, 9,tNow)) * ImOmega + ne9
          pdf(IJKL_W(i+1,j  ,k+1,10,tNext)) = pdf(IJKL_R(i,j,k,10,tNow)) * ImOmega + ne10
          pdf(IJKL_W(i  ,j+1,k+1,11,tNext)) = pdf(IJKL_R(i,j,k,11,tNow)) * ImOmega + ne11
          pdf(IJKL_W(i-1,j  ,k+1,12,tNext)) = pdf(IJKL_R(i,j,k,12,tNow)) * ImOmega + ne12
          pdf(IJKL_W(i  ,j-1,k+1,13,tNext)) = pdf(IJKL_R(i,j,k,13,tNow)) * ImOmega + ne13
          pdf(IJKL_W(i  ,j  ,k-1,14,tNext)) = pdf(IJKL_R(i,j,k,14,tNow)) * ImOmega + ne14
          pdf(IJKL_W(i+1,j  ,k-1,15,tNext)) = pdf(IJKL_R(i,j,k,15,tNow)) * ImOmega + ne15
          pdf(IJKL_W(i  ,j+1,k-1,16,tNext)) = pdf(IJKL_R(i,j,k,16,tNow)) * ImOmega + ne16
          pdf(IJKL_W(i-1,j  ,k-1,17,tNext)) = pdf(IJKL_R(i,j,k,17,tNow)) * ImOmega + ne17
          pdf(IJKL_W(i  ,j-1,k-1,18,tNext)) = pdf(IJKL_R(i,j,k,18,tNow)) * ImOmega + ne18
        end if
      enddo
    enddo
  enddo
end subroutine
```

D.2 MemPHis Library

This section presents the C source code of the *MemPHis* library, which is the Memory Visualizer. It works with a buffer which stores image data and is allocated by the calling program. For that, helper routines exist: `mv_getsize()` returns the required size in bytes for buffer that has to be allocated. `mv_init()` initializes the allocated buffer. `storeIntToBuffer()` and `loadIntFromBuffer()` are internally used functions. `mv_reset()` resets whole image to white and `mv_flush()` dumps image data to external bitmap file. Due to limited space in the following only `mv_step_db()` is printed, which returns the according `double` of user's array and marks the corresponding pixel in picture. The also provided `mv_step_int()` has the same functionality for integer based data and is therefore not printed here. For being able to link this library into a Fortran program without any problems, all data is passed by pointers and all routines' names end with an underbar ('_').

For detailed description of *MemPHis* library see Appendix C.

Algorithm D.2: MemPHis: Memory Visualizer Library

```
#include <stdarg.h>
#include <time.h>
#include <stdio.h>
#include <sys/timeb.h>

/************************ Buffer **************************/
/*                                                        */
/*  Header:                                               */
/*   0..3:  (4 Bytes) Length of Buffer (in Bytes)         */
/*   4..7:  (4 Bytes) Length of Header (Offset for Image-Data)*/
/*   8..11: (4 Bytes) Cachelinesize                       */
/*  12..15: (4 Bytes) Type of Array-Data (i.e. Number of  */
/*                    Bytes Datatype needs)               */
/*  16..19: (4 Bytes) Length of Begins- and Ends- Arrays  */
/*                    (Number of legal elements!)         */
/*  20..??: (LengthofBeginArray Bytes) Begins-Array (int) */
/*                                  (only legal elements!) */
/*  ??..??: (LengthofEndsArray Bytes) Ends-Array (int)    */
/*                                 (only legal elements!) */
/*  ??..??: (2 Bytes) reserved (empty)                    */
/*                                                        */
/*  Body (Image-Data)                                     */
/*  ImageDataOffset..sizeOfBuffer: 3 Bytes per Image-Pixel */
/*                                representing RGB        */
/*                                                        */
/**********************************************************/

#define bufHeaderConstSize 20
#define bufaddrBufferLength 0
#define bufaddrHeaderLength 4
#define bufaddrCachelinesize 8
#define bufaddrArrayType 12
#define bufaddrBeginsEndsLength 16
#define bufaddrBegins 20


void storeIntToBuffer ( unsigned char* buffer, int bytePos, int value ) {
  // Stores value in buffer at bytePos
  ((int*)buffer)[bytePos>>2]=value;
}

int loadIntFromBuffer ( unsigned char* buffer, int bytePos ) {
  // Loads 4-Byte Integer from buffer, starting with bytePos
  // bytepos has to be an integer multiple of 4 to get valid value
  return ((int*)buffer)[bytePos>>2];
}

int mv_getsize_ ( int* arraysize, int* datatype, int* descrsize ) {
  // returns memory size needed by buffer that user has to allocate
  return 3*(*arraysize / *datatype) +   // Body-Data
    bufHeaderConstSize + 2*(*descrsize);  // Header-Data
}

int mv_init_ ( unsigned char* buffer, int* buffersize, int* arraysize, int* cachelinesize,
      int* datatype, int* Begins, int* Ends) {
  // initializes header of buffer and sets all Image-Pixels to white
  // Returns: 0 if all is OK
  //         -1 if buffer size is not right
  //         -2 if Begins- and Ends- Arrays have different lengths
  //         -3 if datatype is not known
  // Parameters:
  //     buffer:         buffer to initialize. will be used by all mv_ functions
  //     buffersize:     size of buffer in Bytes
  //     arraysize:      size of user's array in Bytes
  //     cachelinesize:  length of a cacheline which is to be simulated (in Bytes)
  //     datatype:       size of user's array's datatype in Bytes
  //     Begins:         pointer to Begins-Array
  //     Ends:           pointer to Ends-Array
  //
  // Remarks:
```

```
  //   - buffer is char* to be able to address each byte directly
  //   - Begins-Array contains lower bounds of user's array,
  //     Ends-Array contains the upper bounds.
  //     fastest (stride 1) Dimension first. Array has to be terminated with a -1
  //

  int SizeOfHeader;
  int SizeOfBegins;
  int SizeOfEnds;
  int i;

  i=0;

  // Get size of Begins and Ends arrays
  while (Begins[i]>-1) i++;
  SizeOfBegins=i;
  i=0;

  while (Ends[i]>-1) i++;
  SizeOfEnds=i;
  if (SizeOfEnds!=SizeOfBegins) return -2;

  // calculate the HeaderSize
  SizeOfHeader= bufHeaderConstSize + ((SizeOfBegins+1)*sizeof(int)) +
                ((SizeOfEnds+1)*sizeof(int));

  // check buffer size
  if (*buffersize != (SizeOfHeader + 3 * (*arraysize / *datatype))) {
    fprintf(stderr,"mv_init: Wrong buffer size! Buffer has %d Bytes. Should have %d\n",*
        buffersize,SizeOfHeader + 3 * (*arraysize / *datatype));
    return -1;
  }

  // check if datatype is supported
  if ((*datatype!=sizeof(double)) && (*datatype!=sizeof(int))) return -3;

  // fill the buffer
  storeIntToBuffer(buffer,bufaddrBufferLength,*buffersize);
  storeIntToBuffer(buffer,bufaddrHeaderLength,SizeOfHeader);
  storeIntToBuffer(buffer,bufaddrCachelinesize,*cachelinesize);
  storeIntToBuffer(buffer,bufaddrArrayType,*datatype);
  storeIntToBuffer(buffer,bufaddrBeginsEndsLength,SizeOfBegins);
  for (i=0;i<SizeOfBegins;i++) {
    storeIntToBuffer(buffer,bufHeaderConstSize+i*sizeof(int),Begins[i]);
    storeIntToBuffer(buffer,bufHeaderConstSize+(SizeOfBegins+i)*sizeof(int),Ends[i]);
  }

  // initialize Image-Data with white pixels
  for (i=SizeOfHeader; i<*buffersize; i+=3) {
    buffer[i]  =(unsigned char) 255;  // Red for Pixel i
    buffer[i+1]=(unsigned char) 255;  // Green for Pixel i
    buffer[i+2]=(unsigned char) 255;  // Blue for Pixel i
  }
  if (i!=*buffersize) return -1;

  return 0;
}

double mv_step_db_(unsigned char* buffer, double* theArray, int* color,  ...) {
  // performs a memory access, returns the requested double from theArray and sets
  // the corresponding pixel to color
  // Format of color:
  //     Byte 0 (MSB): reserved
  //     Byte 1      : Red component
  //     Byte 2      : Green component
  //     Byte 3 (LSB): Blue component
  //          example purple: color = 0xFF00FF
  //
  //
  // after color the indices of requested element of theArray must be specified
  // according to order of Begins- and Ends-Array the stride1-Dimension first
  //

  va_list vl;
  int i;
  int address;
  int NumberOfDimensions;
  int bufferDataOffset;
  int dimfactor;
  unsigned char red;
  unsigned char green;
  unsigned char blue;

  va_start( vl, color ); // Initialize variable arguments

  // We assume that there are exactly as much parameters as we need
  // We need as much parameters as Begins and Ends entries have
  address=0;
  NumberOfDimensions=loadIntFromBuffer(buffer,bufaddrBeginsEndsLength);

  dimfactor=1;

  address += *(va_arg(vl, int*));

  for( i = 0; i<NumberOfDimensions-1; i++ )    {
    dimfactor*=(loadIntFromBuffer(buffer,bufaddrBegins+(NumberOfDimensions+i)*sizeof(int))
        - loadIntFromBuffer(buffer,bufaddrBegins+i*sizeof(int))+1);
    address += *(va_arg(vl, int*)) * dimfactor;
  }
  va_end( vl );
  bufferDataOffset=loadIntFromBuffer(buffer,bufaddrHeaderLength);

  // extract color-Information
  blue = *color & 0x000000FF;           // lowest 8 Bit for blue
  green = (*color & 0x0000FF00) >> 8;   // second 8 Bit for green
  red = (*color & 0x00FF0000) >> 16;    // third 8 Bit for red

  buffer[bufferDataOffset+(3*address)]   = red;
  buffer[bufferDataOffset+(3*address)+1] = green;
  buffer[bufferDataOffset+(3*address)+2] = blue;

  return theArray[address]; // return user's requested data
}

void mv_reset_(unsigned char* buffer) {
  // resets image to white
  int i;
  for (i=loadIntFromBuffer(buffer,bufaddrHeaderLength); i<loadIntFromBuffer(buffer,
      bufaddrBufferLength); i+=3) {
    buffer[i]  =(unsigned char) 255;  // Red for Pixel i
    buffer[i+1]=(unsigned char) 255;  // Green for Pixel i
    buffer[i+2]=(unsigned char) 255;  // Blue for Pixel i
  }
}

void mv_flush_(unsigned char* buffer) {
  // writes image to file
  // filename is generated by using current system time

  time_t TheTime;
  struct timeb Time2;
  char Filename[48];
  FILE* fh;
  int FileSize;
  int bufferHeaderLength;
  int bufferBufferLength;
  int i;
  int k;
  int ImageWidth;
  int ImageHeight;
  int tempint;

  TheTime=time(NULL);
  ftime (&Time2);
  sprintf(Filename,"MemPHis_out_%d.%d.bmp",(int)TheTime,Time2.millitm);

  fh=fopen(Filename,"w+");
  if (fh==NULL) {
    fprintf(stderr,"Failure opening image file %s\n", Filename);
    return;
  }

  bufferHeaderLength=loadIntFromBuffer(buffer,bufaddrHeaderLength);
  bufferBufferLength=loadIntFromBuffer(buffer,bufaddrBufferLength);

  /* Write Header in image file */
  fprintf(fh,"BM");
  tempint=0;
  fwrite(&tempint,sizeof(int),1,fh); // we don't know the FileSize yet. Therefore we write
      first 4 zeros
  fwrite(&tempint,sizeof(int),1,fh); // Write reserved Bytes (=0000)

  tempint=54;
  fwrite(&tempint,sizeof(int),1,fh); // Write Offset to RasterData

  /* Write Info-Header in image-File */
  tempint=40;
  fwrite(&tempint,sizeof(int),1,fh); // Write Size of InfoHeader
 ImageWidth=loadIntFromBuffer(buffer,bufaddrCachelinesize)/loadIntFromBuffer(buffer,
     bufaddrArrayType);
  fwrite(&ImageWidth,sizeof(int),1,fh);
  ImageHeight=(loadIntFromBuffer(buffer,bufaddrArrayType)*(bufferBufferLength-
      bufferHeaderLength))/(3*loadIntFromBuffer(buffer,bufaddrCachelinesize));
  if ((loadIntFromBuffer(buffer,bufaddrArrayType)*(bufferBufferLength-bufferHeaderLength))
      %(3*loadIntFromBuffer(buffer,bufaddrCachelinesize))!=0) {
    ImageHeight++;
  }
  fwrite(&ImageHeight,sizeof(int),1,fh);

  // Write Number of Planes (=1, 2 Bytes) and Bits per Pixel (=24 for 24Bit, 2 Bytes)
  // (LSB) 0x01, 0x00, 0x18, 0x00
  tempint=1572865; // 0x00180001
  fwrite(&tempint,sizeof(int),1,fh);

  tempint=0;
  fwrite(&tempint,sizeof(int),1,fh); // Write Type of Compression (no Compression = 0000)
  fwrite(&tempint,sizeof(int),1,fh); // Write compressed size (=0000)

  tempint=2835;  // 0x00000B13
  fwrite(&tempint,sizeof(int),1,fh); // Write horiz resolution
  fwrite(&tempint,sizeof(int),1,fh); // Write vert resolution

  tempint=0;
  fwrite(&tempint,sizeof(int),1,fh); // Write Numbers of Colors (=0000)
  fwrite(&tempint,sizeof(int),1,fh); // Write Colors important (=0000)

  k=0;
  /* Write Image-Pixel-Data */
  for (i=bufferHeaderLength;i<bufferBufferLength;i+=3) {
    fputc(buffer[i],fh);
    fputc(buffer[i+1],fh);
    fputc(buffer[i+2],fh);
    k++;
    // zeropadding
    if ((k%ImageWidth == 0) && ((k*3)%4 != 0)) {
      for (k = (k*3)%4; k<4; k++) {
        fputc(0,fh);
      }
      k=0;
    }
  }

  if (((bufferBufferLength-bufferHeaderLength)%(3*ImageWidth))!=0) {
    // last line of Image has not ImageWidth pixels yet (because bufferData stores no
        integer multiplicative of cachelinesize)
    // -> so we have to fill this last line with enough pixels and pad it with zeros if
        necessary
    for (i=0; i<(ImageWidth-(((bufferBufferLength-bufferHeaderLength)/3)%ImageWidth)); i++)
        {
```

```
      fputc(255,fh);
      fputc(255,fh);
      fputc(255,fh);
      k++;
      if ((k%ImageWidth==0) && ((k*3)%4 != 0)){
        for (k = (k*3)%4; k<4; k++) {
          fputc(0,fh);
        }
      }
    }
  }
  /* Write FileSize */
  FileSize=ftell(fh); // getting current number of written bytes

  fseek(fh,2,SEEK_SET); // moving to third Byte in File (Position of FileSize)
  fwrite(&FileSize,sizeof(int),1,fh); // Write FileSize

  if( fclose( fh ) )
    fprintf(stderr, "The␣image␣file␣%s␣was␣not␣closed\n", Filename );
}
```

Appendix E

Architectures

Following the key features of the platforms that were used for benchmarking.

E.1 Platform 1

- Name: IA32-Cluster (Node)
- CPU: 2x Intel Xeon, 2.66 GHz
- Arithmetic Power: 2 FLOPs/cycle, 5.3 GFLOP/s
- Memory interface: 1 load + 1 store per cycle
- L1 Cache: 8 KB, 4-way set associative, write-through, 64 Bytes per cache line
- L2 Cache: 512 KB, 8-way set associative, write-back, 64 Bytes per cache line, 2 loaded lines per load
- Memory: 2.0 GB, DualChannel-DDR RAM, 133 MHz, 4.3 GB/s

E.2 Platform 2

- Name: Gollum
- CPU: 1x Intel Pentium 4, 3.0 GHz
- Arithmetic Power: 2 FLOPs/cycle, 6.0 GFLOP/s
- Memory interface: 1 load + 1 store per cycle
- L1 Cache: 8 KB, 4-way set associative, write-through, 64 Bytes per cache line
- L2 Cache: 512 KB, 8-way set associative, write-back, 64 Bytes per cache line, 2 loaded lines per load
- Memory: 1.0 GB, DualChannel-DDR RAM, 200 MHz, 6.4 GB/s

E.3 Platform 3

- Name: MC2
- CPU: 2x Intel Itanium 2, 1.4 GHz
- Arithmetic Power: 4 FLOPs/cycle, 5.6 GFLOP/s
- Memory interface: (2 loads + 2 stores) per cycle or 4 loads per cycle
- L1 Cache: only for Integer data
- L2 Cache: 256 KB, 8-way set associative, write-back, 128 Bytes per cache line
- L3 Cache: 1.5 MB, 6-way set associative, write-back, 128 Bytes per cache line
- Memory: 10.0 GB, 256 bit bus width, 200 MHz (FSB 400), 6.4 GB/s

E.4 Platform 4

- Name: Altix
- CPU: 28x Intel Itanium 2, 1.3 GHz
- Arithmetic Power: 4 FLOPs/cycle, 5.2 GFLOP/s
- Memory interface: (2 loads + 2 stores) per cycle cycle or 4 loads per cycle
- L1 Cache: only for Integer data
- L2 Cache: 256 KB, 8-way set associative, write-back, 128 Bytes per cache line
- L3 Cache: 3.0 MB, 12-way set associative, write-back, 128 Bytes per cache line
- Memory: DSM machine, 112.0 GB distributed shared memory with NUMAlink technology. On one C-Brick there are 2 processors with 6.4 GB/s to SHUB which connects them to 8 GB memory (10.2 GB/s) and another SHUB with two processors (2x 3.2 GB/s). 7 C-Bricks are interconnected by buses with 4x 3.2 GB/s.

E.5 Platform 5

- Name: Thor
- CPU: 2x AMD Opteron 1.6 GHz,
- Arithmetic Power: 2 FLOPs/cycle, 3.2 GFLOP/s
- Memory interface: (1 load + 1 store) per cycle or 2 loads per cycle
- L1 Cache: 64 KB, 2-way set associative, write-back
- L2 Cache: 2.0 MB, 16-way set associative, write-back
- Memory: 6.0 GB, DualChannel-DDR RAM, 266 MHz , 4.3 GB/s

Appendix F

Compiler

For compiling *LBMKernel* on different test systems the following compilers in version and build were used with specified flags.

- Intel 32 Bit
  - Version 7.1
    - Compiler: Intel(R) Fortran Compiler for 32-bit applications, Version 7.1, Build 20030730Z
    - Linker: GNU ld version 2.13.90.0.18 20030121 Debian GNU/Linux
    - Options:
      - Compiler: `ifc -O3 -xW -tpp7 -fno-alias`
      - Linker: `ifc -O3 -xW -tpp7 -fno-alias -static`
    - Directives:
      - `!DEC$ VECTOR NONTEMPORAL`
      - `!DEC$ VECTOR ALIGNED`
      - `!DEC$ IVDEP`
  - Version 8.0
    - Compiler: Intel(R) Fortran Compiler for 32-bit applications, Version 8.0, Build 20040412Z, Package ID l_fc_pc_8.0.046
    - Linker: GNU ld version 2.13.90.0.18 20030121 Debian GNU/Linux
    - Options:
      - Compiler: `ifort -O3 -xW -tpp7 -fno-alias`
      - Linker: `ifort -O3 -xW -tpp7 -fno-alias -static`
    - Directives:
      - `!DEC$ VECTOR NONTEMPORAL`
      - `!DEC$ VECTOR ALIGNED`
      - `!DEC$ IVDEP`
- Intel 64 Bit
  - Version 7.1
    - Compiler: Fortran Itanium(R) compiler, Version 7.1, Build 20030909
    - Linker: GNU ld version 2.11.90.0.8 (with BFD 2.11.90.0.8)
    - Options:
      - Compiler: `efc -O3 -ivdep_parallel -fno-alias`

      - Linker: `efc -O3 -ivdep_parallel -fno-alias -static`
    - Directives:
      - `!DEC$ IVDEP`
  - Version 8.0
    - Compiler: Fortran Itanium(R) compiler, Version 8.0,
      Build 20040416, Package ID l_fc_pc_8.0.046
    - Linker: GNU ld version 2.11.90.0.8 (with BFD 2.11.90.0.8)
    - Options:
      - Compiler: `ifort -O3 -ivdep_parallel -fno-alias`
      - Linker: `ifort -O3 -ivdep_parallel -fno-alias -static`
    - Directives:
      - `!DEC$ IVDEP`

- Portland Group 64 Bit
  - Version 5.1-3
    - Compiler: pgf90 5.1-3
    - Options: `pgf90 -O4 -tp k8-64 -fastsse -Mpreprocess -Mnontemporal`
  - Version 5.0-2
    - Compiler: pgf90 5.0-2
    - Options: `pgf90 -O4 -tp k8-64 -fastsse -Mpreprocess -Mnontemporal`

Bibliography

[AMD99] *Implementation of Write Allocate in the K86 Processors*. Whitepaper No. 21326. Advanced Micro Devices, Inc. February 1999.

[CDo98] S. Chen, G. D. Doolen. *Lattice Boltzmann Method for Fluid Flows*. Annual Review of Fluid Mechanics. 1998.

[CDr98] B. Chopard, M. Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press, Collection Aléa-Saclay: Monographs and Texts in Statistical Physics, Cambridge, United Kingdom. 1998.

[Dau98] J. Daub. Description of BMP file format. Jörn Daub EDV Beratung, Glashütter Weg 105, 22889 Tangstedt, Germany. 14.01.1998. http://www.daubnet.com/formats/BMP.html (last visited on 25.07.2004).

[FPe96] J. H. Ferziger, M. Perić. *Computational Methods for Fluid Dynamics*. Springer, Berlin, Heidelberg, New York. 1996.

[GHo01] S. Goedecker, A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. siam, Society of Industrial Applied Mathematics, Philadelphia, USA. 2001.

[Han98] J. Handy. *The Cache Memory Book*. Second edition. Academic Press, Inc. 1998.

[Igl03] K. Iglberger. *Cache Optimizations for the Lattice Boltzmann Method in 3D*. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Cauerstraße 6, D-91058 Erlangen, Germany. August 2003.

[Int99] *IA-32 Intel® Architecture Optimization*, Reference Manual. Document Number: 248966-011. Intel Corporation. 1999-2004.

[Int00] *Intel® VTune Performance Analyzer - Documentation*, Reference. Intel Corporation. 2000-2003.

[Int03] *Intel® Fortran Compiler for Linux Systems*, User's Guide, Volume II: Optimizing Applications. Document Number: 253260-001. Intel Corporation. 2003-2004.

[MSY02] Renwei Mei, Wei Shyy, Dazhi Yu, Li-Shi Luo. *Lattice Boltzmann Method for 3-D Flows with Curved Boundary*. NASA/CR-2002-211657. NASA Center for AeroSpace Information (CASI), 7121 Standard Drive, Hanover, MD 21076-1320, June 2002.

[Pra01] Gurpur M. Prabhu. *Computer Architecture Tutorial*. Department of Computer Science, Iowa State University, Ames, IA 50011, USA. 2001. http://www.cs.iastate.edu/~prabhu/Tutorial/title.html (last visited on 24.06.2004).

[PTD04] T. Pohl, N. Thürey, F. Deserno, U. Rüde, P. Lammers. *Parallel Performance of Large-Scale Lattice Boltzmann Applications.* Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Cauerstraße 6, D-91058 Erlangen, Germany. April 2004.

[QHL92] Y. H. Qian, D. d'Humières, P. Lallemand. *Lattice BGK Models for Navier-Stokes Equation.* Europhysics Letters. 1992.

[Sch00] W. Schönauer. *Scientific Supercomputing*, Architecture and Use of Shared and Distributed Memory Parallel Computers. Willi Schönauer, Wilhelm-Kolb-Straße 5b, D-76187 Karlsruhe, Germany. 2000.

[Wei03] Software *Calltree* in combination with visualization tool *KCachegrind.* Developed by J. Weidendorfer from November 2003 until today. `http://kcachegrind.sourceforge.net` (last visited on 28.06.2004).

[Wei04] Correspondence with J. Weidendorfer, developer of *kcachegrind* [Wei03], by E-Mail on 30.06.2004.

[WoG00] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models.* Springer, 2000.

[YML01] Dazhi Yu, Renwei Mei, Li-Shi Luo, Wei Shyy. *Viscous flow computations with the method of lattice Boltzmann equation.* Progress in Aerospace Sciences. Elsevier Science Ltd. 2003.

[ZWH04] T. Zeiser, G. Wellein, G. Hager, S. Donath, F. Deserno, P. Lammers, M. Wierse. *Optimized Lattice Boltzmann Kernels as Testbeds for Processor Performance.* Regional Computing Center of Erlangen (RRZE), Martensstraße 1, D-91058 Erlangen, Germany. April 2004.