Implementation and Optimization of the Lattice Boltzmann Method for the Jackal DSM System

Bachelor Thesis im Fach Mathematik

vorgelegt von

Alexander Dreweke

geb. 08. Januar 1980 in Ingolstadt

angefertigt am

Institut für Informatik
Lehrstuhl für Informatik 10
Systemsimulation
Friedrich–Alexander–Universität Erlangen–Nürnberg
(Prof. Dr. Ulrich Rüde)

in Zusammenarbeit mit

Institut für Informatik
Lehrstuhl für Informatik 2
Programmiersysteme
Friedrich–Alexander–Universität Erlangen–Nürnberg
(Prof. Dr. Michael Philippsen)

Betreuer: Dipl.–Inf Michael Klemm
Prof. Dr. Ulrich Rüde
Prof. Dr. Michael Philippsen

Beginn der Arbeit: 16.06.2005
Abgabe der Arbeit: 04.08.2005

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 04.08.2005

Alexander Dreweke

Abstract

The goal of this thesis is to implement and optimize the lattice–Boltzmann method for the distributed shared memory system Jackal.

The lattice Boltzmann method is a new and promising alternative to the traditional Navier–Stokes solvers in the domain of computational fluid dynamics. It is based on cellular automata and operates on a domain, were all cells are normalized in space. Because of the great need for memory and computing time most computational fluid dynamic problems cannot be solved on a single computer in sufficient time. Therefore the solvers have to be parallelised. Because of the implicit parallelism of the lattice Boltzmann method this promises a good speed–up.

For parallelization a distributed shared memory approach is instead of the traditional Message Passing Interface approach. The distributed shared memory system provides the programmer a global address space on top of the distributed memory of the individual nodes of the cluster. Hence, all data can be addressed regardless of which node allocated them. The Jackal system implements such a distributed shared memory system for Java programs by cooperation of it's compiler and runtime system. The Jackal system is also capable of automatically distribute a multi–threaded program on a cluster.

Our performance evaluation shows that by optimizing the lattice Boltzmann method the sequential program we are able to achieve 5.5 millions of lattice updates per second for a 1000×1000 grid. By further optimizing the sequential program in a distributed shared memory specific way we are able to achieve a speed–up of 3.4 in both 2D and 3D test case. This results in a total of 2 millions of lattice updates per second.

Contents

1 Introduction 1

1.1 Lattice Boltzmann Method . . . 1
1.2 The Jackal System . . . 1
1.3 Contents . . . 2

2 Introduction to the Lattice Boltzmann Method 3

2.1 The Lattice Boltzmann Method . . . 3
2.2 The Lattice Boltzmann Equations . . . 3
2.3 Lid–driven Cavity . . . 6
2.4 Boundary Conditions . . . 6
2.5 Lattice Boltzmann Models . . . 8
2.5.1 Two–dimensional Model . . . 8
2.5.2 Three–dimensional Model . . . 10
2.6 Summary . . . 11

3 Jackal 13

3.1 The Jackal Compiler . . . 13
3.2 The Jackal Runtime System . . . 17
3.3 Architecture . . . 19
3.3.1 Memory Model . . . 20
3.3.2 DSM Optimizations . . . 22
3.4 Summary . . . 23

4 Implementation and Optimization 25

4.1 Data Structures . . . 25
4.1.1 Two–dimensional Data Model . . . 26
4.1.2 Three–dimensional Data Model . . . 28
4.2 Sequential Program Optimizations . . . 29
4.3 DSM–specific Optimizations . . . 32

4.3.1 Thread–parallel Memory–allocation 32
4.3.2 Association of Boundary Cells 35
4.4 Summary 37

5 Performance Evaluation 39
5.1 Two–dimensional Testcase 39
5.2 Three–dimensional Testcase 42
5.3 Summary 46

6 Related and Future Work 49
6.1 Related Work 49
6.2 Future Work 50

7 Summary 51

A Commandline Options 53

List of Figures

2.1 Streaming step: particles from surrounding cells flow into the cell. 4
2.2 Collide step: existing and inflowed particles are weighted. 4
2.3 Velocity field with lid–driven cavity in a 25×25 domain after 10000 time steps. 7
2.4 Velocity field with lid–driven cavity in a $25 \times 25 \times 10$ domain after 10000 time steps. 7
2.5 2D domain with ghost cells. 8

3.1 Overview over the compiler pipeline of the Jackal Compiler. 14
3.2 Translation from Java code to LASM with access check. 15
3.3 Optimization: Object inlining 16
3.4 Optimization: Method inlining. 18
3.5 Architectural view of the Jackal runtime system. 19
3.6 High–level view on the DSM provided by the Jackal runtime system. . . 21

4.1 Interface of the Grid2D class. 26
4.2 Memory layout of the 2D array data structure. 27
4.3 Memory Layout of the 2D vector data structure. 27
4.4 Memory Layout of the 3D array data structure. 28
4.5 Memory Layout of the 3D vector data structure. 29
4.6 An unoptimized 2D LBM core. 30
4.7 2D LBM core with inlined data structure accesses. 31
4.8 2D LBM core with unrolled innermost loop. 33
4.9 2D LBM core with blocking. 34
4.10 2D Domain: with 9 subdomains. 36

5.1 Runtimes of the benchmarks for the 2D case with different compilers. . . 41
5.2 MLU/s of the benchmarks for the 3D case with different compilers. . . . 41
5.3 Runtimes of the DSM benchmarks for the 2D case with different optimizations. 43
5.4 Runtimes of the benchmarks for the 3D case with different compilers. . . 45
5.5 MLU/s of the benchmarks for the 3D case with different compilers. . . . 45

5.6 Runtimes of the DSM benchmarks for the 3D case with different optimizations. 47

List of Tables

5.1 Test setup data. 39
5.2 Compiler setup data. 40
5.3 Runtimes of the benchmarks for the 2D case with different compilers. . . 40
5.4 MLU/s of the benchmarks for the 2D case with different compilers. . . . 40
5.5 Runtimes of the DSM benchmarks for the 2D case with different optimizations. 42
5.6 Runtimes of the benchmarks for the 3D case with different compilers. . . 44
5.7 MLU/s of the benchmarks for the 3D case with different compilers. . . . 44
5.8 Runtimes of the DSM benchmarks for the 3D case with different optimizations. 46

1 Introduction

The main goals for this thesis were to implement a fluid dynamics simulation in 2D and 3D based on the lattice Boltzmann method (LBM). This implementation should then be optimized using standard optimizations techniques. Then the sequential implementation should be parallelized by using threads. This multi–threaded version then will be optimized for the use with the Distributed Shared Memory (DSM) system Jackal.

1.1 Lattice Boltzmann Method

The lattice Boltzmann method is based on cellular automata and operates with normalized time steps on a domain where all cells are normalized in space. Each of these cells contains a finite number of discrete states (the so–called distribution functions). These states are updated synchronously at each discrete time step in all cells of the domain.

The lattice Boltzmann method therefore is a new and promising alternative to the traditional Navier–Stokes solvers in the domain of Computational Fluid Dynamics (CFD). Because of the great need for memory and computing time most CFD problems can not be solved on a single computer in sufficient time, therefore the solvers have to be parallelized. Because of the implicit parallelism of the lattice Boltzmann method this promises a good speed–up. For parallelization a DSM approach was chosen in this thesis instead of Message Passing Interface (MPI) approach.

1.2 The Jackal System

The Jackal compiler and the Jackal runtime system (RTS) together implement a shared memory for Jackal programs on distributed physical memory. In a DSM system the programmer does not have to send messages between the various threads or processes to transfer the data from one node to another (unlike by MPI). The DSM system provides a global address space between all nodes where all objects can be addressed regardless of which node allocated them. The Jackal system simulates such a DSM system for Java programs through cooperation of it's compiler and runtime system. The Jackal system is capable to automatically distribute a multi–threaded program in among all nodes in

a cluster.

The Jackal compiler transparently inserts access checks before every object access during the compilation of the program. The runtime system then assures that each object that is accessed on a node is locally available. If the object was not created on the accessing node it is transferred by the RTS to the caching heap of the node.

1.3 Contents

In Chapter 2 the most important lattice Boltzmann equations will be derived. In addition to the various equations the 2D and 3D lattice Boltzmann models will be described. Chapter 3 will give an overview about the Jackal system and will explain how the software–based distributed–shared memory system is build through the cooperation of the various parts of the Jackal framework.

After the theoretical concepts have been shown the implementation of the LBM with the Jackal framework will be explained in Chapter 4. Various optimization techniques for the sequential LBM and the thread parallization will be shown. In Chapter 5 then measurements will proof the speedup of the optimizations that have been used.

Chapter 6 will give an overview of related projects in distributed cluster computing and will furthermore describe some additional features which could extend the performance of Jackal parallized programs. Chapter 7 will conclude the thesis.

2 Introduction to the Lattice Boltzmann Method

In this chapter a short introduction into the lattice Boltzmann method (LBM) is given. At first the major steps of the derivation of the equations of the LBM are outlined, followed by a short description of the lid driven–cavity testcase (see section 2.3) and the boundary conditions used for the implementation. At last a description of lattice models *D2Q9* (see section 2.5.1) and *D3Q19* (see section 2.5.2) is given.

2.1 The Lattice Boltzmann Method

The lattice Boltzmann method [30] represents an alternative approach, based on cellular automata, to the traditional solvers that are based on the Navier–Stokes equations [14], for the direct simulation of the flow of incompressible fluids. Cellular automata simulate a physical system in which space and time are discretized. The whole domain if build of cells. Each of these cells holds a finite number of discrete states called distribution functions. These states are updated synchronously at each discrete time step. The update rules are deterministic and uniform in the whole discrete domain. In addition the evolution of a cell only depends on the finite states in the neighboring cells.

2.2 The Lattice Boltzmann Equations

During a single discrete time step the particles in each cell of the domain stream into the corresponding neighboring cells (the so–called *stream step*) according to their velocity vector, as can be seen in Figure 2.1. In each neighbor cell the particles collide with the other particles from the other surrounding cells (the so–called *collide step*), see Figure 2.2. The result of these collisions is calculated by solving the kinetic Boltzmann equation for the particle distribution in that region. Afterwards this result is used as the new particle distribution for the *stream step* of the next discrete time step of this cell.

We will now outline the major steps of the derivation of the relevant equations of the lattice Boltzmann method. The Boltzmann equation—with external forces neglected—

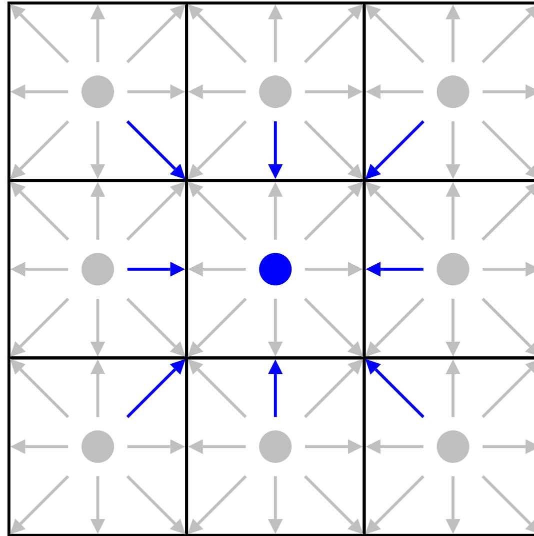


Figure 2.1: Streaming step: particles from surrounding cells flow into the cell.

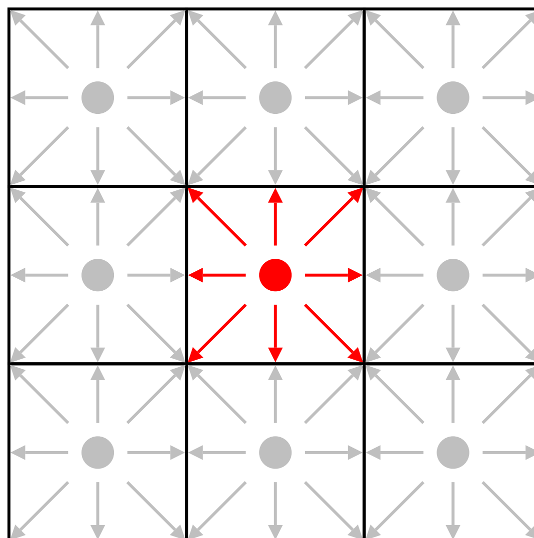


Figure 2.2: Collide step: existing and inflowed particles are weighted.

can be written as:

$$\frac{\delta f}{\delta t} + \xi \cdot \frac{\delta f}{\delta x} = Q(f, f). \tag{2.1}$$

$Q(f, f)$ consists of a complex integrodifferential expression. Using the Bhatnager–Gross–Krook (BGK) model [3] the integral can be simplified for low Mach numbers

$$Q(f, f) = -\frac{1}{\tau}\left[f - f^{(0)}\right]. \tag{2.2}$$

The Boltzmann equation with a single relaxation time approximation can therefore be constructed using Equation 2.1 and the BGK approximation in Equation 2.2 which leads to Equation 2.3:

$$\frac{\delta f}{\delta t} + \xi \cdot \frac{\delta f}{\delta x} = -\frac{1}{\tau}\left[f - f^{(0)}\right], \tag{2.3}$$

where ξ is the particle velocity, x is the site of the cell in the domain and t is time. The equilibrium distribution function $f^{(0)}$ (Maxwell–Boltzmann distribution function) is used to calculate the values of the particle distribution.

In order to solve the Boltzmann equation numerically, at first Equation 2.3 must be discretized by a finite number of velocities e_i in velocity space ξ which leads to

$$\frac{\delta f_i}{\delta t} + e_i \cdot \frac{\delta f_i}{\delta x} = -\frac{1}{\tau}\left[f_i - f_i^{(0)}\right]. \tag{2.4}$$

For non–thermal fluids a Taylor series expansion can be used to approximate the equilibrium distribution function (Equation 2.4) [17] into

$$f_i = \rho w_i \left[1 + \frac{3}{c^2}\vec{e_i} \cdot u + \frac{9}{2c^4}(\vec{e_i} \cdot u)^2 - \frac{3}{2c^2} \cdot u^2\right], \tag{2.5}$$

where $c = \frac{\Delta x}{\Delta t}$, w_i is a weighting factor, and $\vec{e_i}$ is a discrete velocity vector. To simplify Equation 2.5 we discretize Δx and Δt such that $c = 1$. Leading to a simplified equilibrium distribution function

$$f_i = \rho w_i \left[1 + 3\vec{e_i} \cdot u + \frac{9}{2}(\vec{e_i} \cdot u)^2 - \frac{3}{2} \cdot u^2\right]. \tag{2.6}$$

The mass density $\rho(x,t)$ is by calculated by adding up the particle distributions for each cell. This leads to the following function:

$$\rho(x,t) = \sum_i f_i(x,t) \tag{2.7}$$

Besides the mass density, the velocity density $p(x,t) \cdot u(x,t)$ of the domain is another important visualization method in the field of computational fluid dynamics. The velocity density of each cell in the domain is calculated by adding up the particle distributions—like for the mass density—with respect to the discrete velocities:

$$\rho(x,t) \cdot u(x,t) = \sum_i e_i \cdot f_i(x,t) \tag{2.8}$$

2.3 Lid–driven Cavity

To be able to test and benchmark the lattice Boltzmann implementations, the lid–driven cavity problem, as described below, was chosen for 2D and 3D problems.

The domain for this test consists of a square in the 2D case and a cube in the 3D case respectively. At the beginning $t = 0$ the fluid is resting. The upper most layer of the domain (the so–called "lid") moves with constant velocity in the same direction. You can think of the lid as an endless plate that lies on the domain and is constantly dragged—tangential to the domain—into the same direction. The movement of the lid causes velocities of the other fluid cells in the domain to move in a circular motion. Figure 2.3 for 2D and Figure 2.4 for 3D demonstrate the general behavior of the fluid.

2.4 Boundary Conditions

Considering the problem we are simulating, only solid boundaries are implemented to delimit the domain. The solid boundary conditions used in the LBM are based on the so–called *bounce back* boundary condition. That is, each particle that collides with a solid border has its velocities v_i reversed ($v_i = -v_i$) and therefore bounces back from the border. This is physically appropriate because solid walls have a sufficient roughness to prevent any fluid motion parallel to the wall [24].

Also, the outermost cells that delimit the domain are marked as boundary cells (these cells are shown light gray in Figure 2.5) with the *bounce back* condition. During their

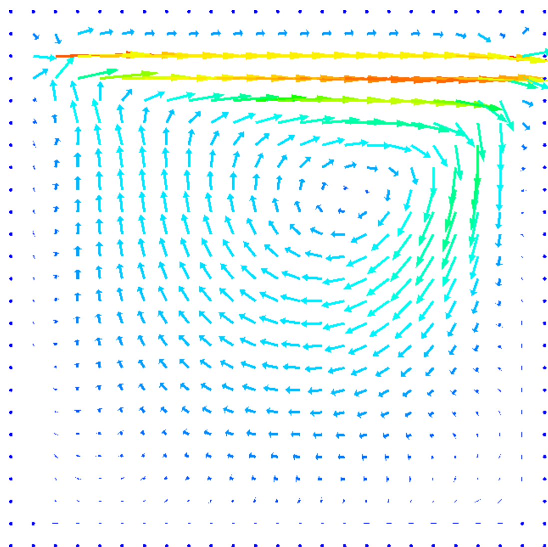


Figure 2.3: Velocity field with lid–driven cavity in a 25×25 domain after 10000 time steps.

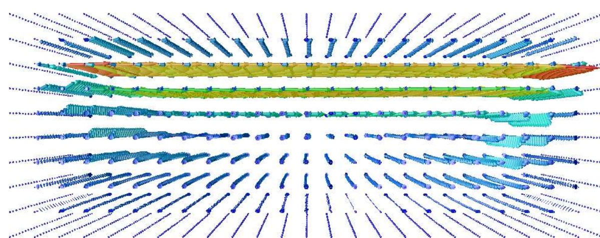


Figure 2.4: Velocity field with lid–driven cavity in a $25 \times 25 \times 10$ domain after 10000 time steps.

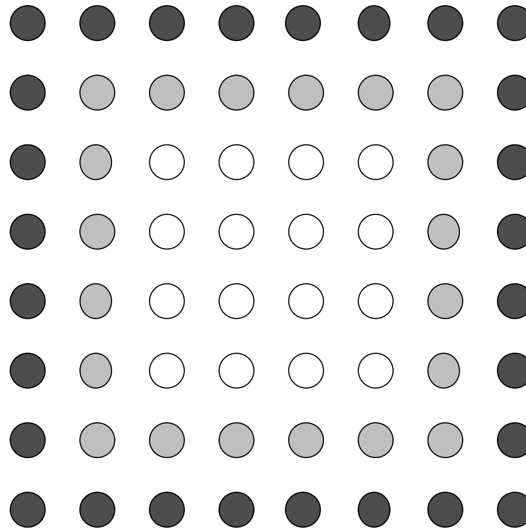


Figure 2.5: 2D domain with ghost cells. Fulid cells are marked white, Boundary cells marked grey and Ghost cells are marked dark.

update, these cells still require access to cells that are outside of the domain (see Figure 2.5), which are the so–called ghost cells (shown dark gray in the figure). To avoid special treatment for those cells another layer of boundary cells is put around the domain.

However, the additional layer of boundary cells are not processed in a lattice Boltzmann step, they only exist to avoid a special treatment for the actual boundary cells.

2.5 Lattice Boltzmann Models

Over time there were various lattice models developed for the LBM. They all follow the same naming scheme DXQY. Where X denotes the number of dimensions of the lattice model and Y specifies the number of discrete velocities in the model. So D2Q9 specifies a two dimensional lattice model with 9 velocities. The following sections will describe the models used in detail.

2.5.1 Two–dimensional Model

Besides the lattice Boltzmann equations that were derived in section 2.5, the distinct velocities and the weighting factors for a certain lattice Boltzmann model are needed.

These missing quantities will be derived in the following sections.

Discrete Velocities

The model of discrete directions of velocity for 2D problems in this thesis is the so called *D2Q9* model, that defines 9 directions. The directions are defined as C (center), E (east), N (north), W (west), S (south), NE (north east), NW (north west), SW (south west) and SE (south east). This set of discrete directions of velocity is defined by the velocity vector $\vec{e_i}$ as follows:

$$\vec{e_i} := \begin{cases} (0,0), & i = C, \\ (\pm 1, 0), & i = E, W, \\ (0, \pm 1), & i = N, S, \\ (\pm 1, \pm 1), & i = NE, NW, SE, SW. \end{cases} \tag{2.9}$$

The vector $\vec{e}_{NW}$ for example would be (1, -1). The components of the velocity vector $\vec{e_i}$ are all normalized to 1 instead of c ($c = \frac{\Delta x}{\Delta t}$) because of the discretization of Δx and Δt we introduced during the derivation of the Boltzmann equation 2.6 in section 2.5.

Weighting factors

The *D2Q9* model includes three different speeds. The weighting factors w_a for directions with identical speeds are equal. Of all the moments of the lattice velocities over w_a the first and third moments vanish. The non–vanishing elements of the moments up to the fourth order read [30]:

- 0. momentum:

$$\sum_a w_a = w_0 + 4w_1 + 4w_2 = \rho_0 \tag{2.10}$$

- 2. momentum:

$$\begin{aligned} \textstyle\sum_a e_{a1}^2 w_a = 2c^2 w_1 + 4c^2 w_2 = \rho_0 \frac{k_B T}{m} \\ \textstyle\sum_a e_{a2}^2 w_a = 2c^2 w_1 + 4c^2 w_2 = \rho_0 \frac{k_B T}{m} \end{aligned} \tag{2.11}$$

- 4. momentum:

$$\begin{aligned} \textstyle\sum_a e_{a1}^4 w_a = 2c^4 w_1 + 4c^4 w_2 = 3\rho_0 (\frac{k_B T}{m})^2 \\ \textstyle\sum_a e_{a2}^4 w_a = 2c^4 w_1 + 4c^4 w_2 = 3\rho_0 (\frac{k_B T}{m})^2 \end{aligned} \tag{2.12}$$

$$\sum_a e_{a1}^2 e_{a2}^2 w_a = 4c^4 w_2 = \rho_0 (\frac{k_B T}{m})^2 \tag{2.13}$$

Now, Equation 2.11 and 2.12 will be used to calculate $\frac{k_B T}{m}$. The result of $\frac{k_B T}{m}$ in combination with Equation 2.12 will result in the weighting factor w_1 and with Equation 2.13 will result in w_2. As Equation 2.10 is the only equation that includes w_0, it will be used to calculate w_0. The solution reads:

$$w_a = f_a(x,0) = \begin{cases} \frac{4}{9}, & a = 0, \\ \frac{1}{9}, & a = 1 \\ \frac{1}{36}, & a = 2. \end{cases} \tag{2.14}$$

2.5.2 Three–dimensional Model

Like for the 2D model (see section 2.5.1) the distinct velocities and the weighting factors must be derived. As far as it is possible, derivations from the 2D model will be re–used to make the approach more understandable.

Discrete Velocities

For 3D problems the model of discrete directions of velocity choosen for this thesis is the so–called *D3Q19* model that defines 19 directions. The directions are identical to the directions of the *D2Q9* model with the following directions T (top), B (bottom), TE (top east), TN (top north), TW (top west), TS (top south), BE (bottom east), BN (bottom north), BW (bottom west) and BS (bottom south) in additon. These 19 discrete directions of velocity are defined by the vector $\vec{e_i}$:

$$\vec{e_i} := \begin{cases} (0,0,0), & i = C, \\ (\pm 1,0,0), & i = E, W, \\ (0,\pm 1,0), & i = N, S, \\ (0,0,\pm 1), & i = T, B, \\ (\pm 1,\pm 1,0), & i = NE, NW, SE, SW, \\ (\pm 1,0,\pm 1), & i = TE, TW, BE, BW, \\ (0,\pm 1,\pm 1), & i = TN, TS, BN, BS. \end{cases} \tag{2.15}$$

The vector $\vec{e}_{TW}$ for example is (-1, 0, 1). As for the 2D model *D2Q9* model the components of the distinct velocity vectors $\vec{e_i}$ are all normalized to 1 because of the

discretization of Δx and Δt during the derivation of the Boltzmann equation 2.6.

Weighting factors

As in the *D2Q9* model the *D3Q19* model for a cubic lattice with 19 distinct velocities includes three different speeds, the weighting factors w_a for directions with identical speeds are equal for reason of symmetry. As in the *D2Q9* model all odd moments vanish and the non–vanishing elements of the moments up to the fourth order read [30]:

- 0. momentum:

$$\sum_a w_a = w_0 + 6w_1 + 12w_2 = \rho_0 \tag{2.16}$$

- 2. momentum:

$$\begin{aligned} \textstyle\sum_a e_{a1}^2 w_a &= 2c^4 w_1 + 8c^4 w_2 = \rho_0 \tfrac{k_B T}{m} \\ \textstyle\sum_a e_{a2}^2 w_a &= 2c^4 w_1 + 8c^4 w_2 = \rho_0 \tfrac{k_B T}{m} \end{aligned} \tag{2.17}$$

- 4. momentum:

$$\begin{aligned} \textstyle\sum_a e_{a1}^4 w_a &= 2c^4 w_1 + 8c^4 w_2 = 3\rho_0 (\tfrac{k_B T}{m})^2 \\ \textstyle\sum_a e_{a2}^4 w_a &= 2c^4 w_1 + 8c^4 w_2 = 3\rho_0 (\tfrac{k_B T}{m})^2 \end{aligned} \tag{2.18}$$

$$\sum_a e_{a1}^2 e_{a2}^2 w_a = 4c^4 w_2 = \rho_0 (\frac{k_B T}{m})^2 \tag{2.19}$$

Now, equation 2.17 and 2.18 will be used to calculate $\frac{k_B T}{m}$. The result of $\frac{k_B T}{m}$ in combination with equation 2.18 will result in the weighting factor w_1 together with equation 2.19 will result in w_2. As equation 2.16 is the only equation that includes w_0, it will be used to calculate w_0. The solution reads:

$$w_a = f_a(x, 0) = \begin{cases} \frac{1}{3}, & a = 0, \\ \frac{1}{18}, & a = 1, \\ \frac{1}{36}, & a = 2. \end{cases} \tag{2.20}$$

2.6 Summary

In this chapter all the main equations of the LBM and all the distinct velocities and the weighting factors for the *D2Q9* and *D3Q19* lattice Boltzmann models were derived. With this knowledge the algorithms that will be shown in Chapter 4 can be easily understood.

3 Jackal

The Jackal compiler and the Jackal runtime system (RTS) together simulate a shared memory on distributed physical memory for Jackal programs.

In order to create a shared memory, the compiler inserts so–called *access checks* before each object access. This is transparently done during the compilation of the Java source code to machine code. With these access checks the Jackal runtime system is then able to establish a software–based distributed shared memory (S–DSM) among all nodes in the cluster.

In the following chapter an overview about the various components of the Jackal system is given and it is explained how these must interact in order to create a S–DSM system in a transparent manner.

3.1 The Jackal Compiler

Jackal is a modular ahead–of–time multi–pass compiler that supports compiling various languages such as Java, C and C++ to native code on different architectures such as Intel IA32, AMD64, Itanium, PowerPC and many more. Currently Jackal is able to create a DSM syste only for programs wirtten in Java.

In traditional approaches Java is translated to an intermediate byte–code representation that can be executed by any Java Virtual Machine (JVM) on all supported computing platforms without the need to recompile the program from source. Because of the rising popularity of Java the need for more efficient execution became necessary. Common appendages to achieve fast execution of Java programs are the use of Just–In–Time (JIT) compilers [6] or the approach of statically compiling byte–code into executable programs without interpreting the byte–code [1, 10].

All these methods share the problem of mapping the stack–machine representation of Java byte–code onto register–based CPUs [31]. Jackal avoids these problems by directly compiling the Java source code to executables. Figure 3.1 shows the whole process in detail. Firstly, the source code is processed by the Java frontend and is transformed to the intermediate code LASM which is then further processed by the compiler's LASM

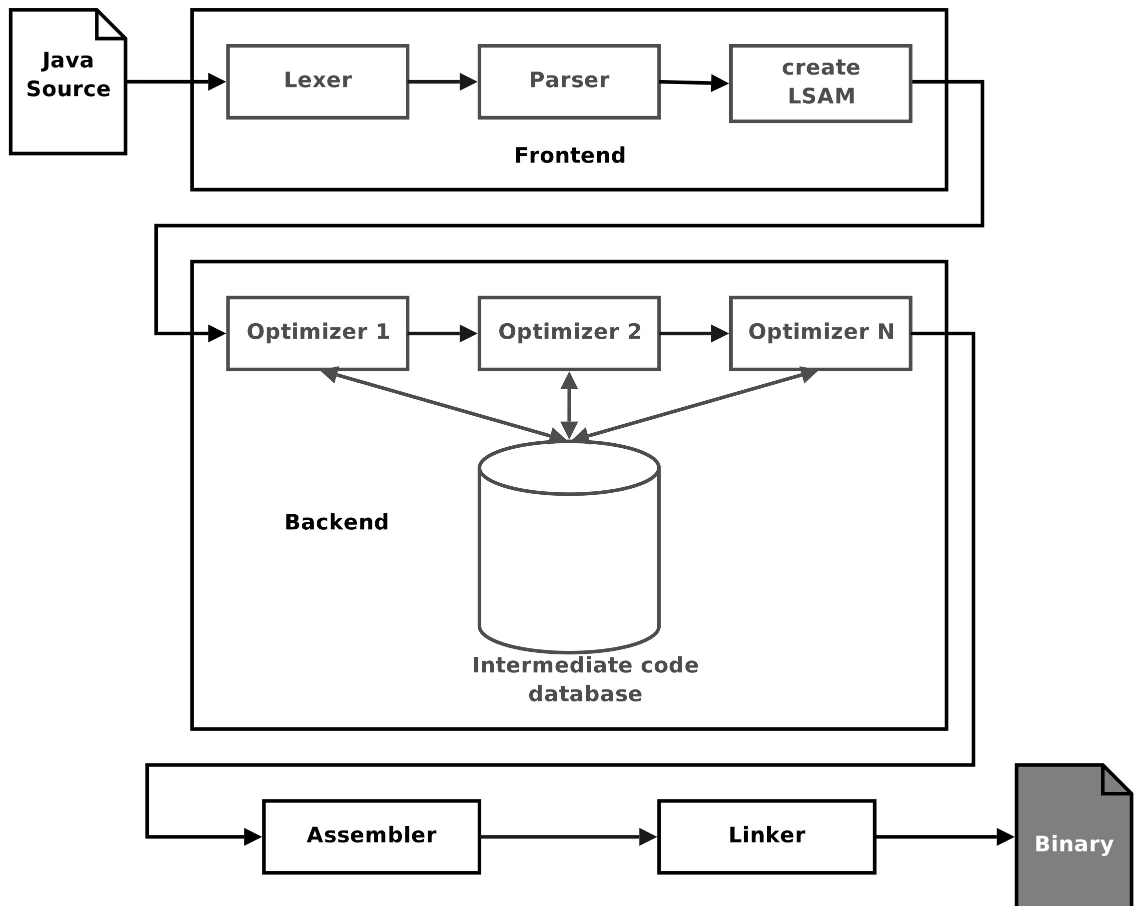


Figure 3.1: Overview over the compiler pipeline of the Jackal Compiler.

backend. After all the LASM passes have been executed on the intermediate code, it is transformed to machine level instructions which are assembled and linked by external tools to the final executable.

Most S–DSM systems analyze the binary code, then they add software access checks by rewriting the binary code, and optimize it afterwards by combining and removing some of the checks again [21, 20, 22]. Jackal's approach is to provide the functionality of an S–DSM system by adding source–level access checks [27]. The compiler's frontend has to add an access check for every object access. After converting high–level LASM to low–level LASM these access checks consist of two parts:

- A conditional jump that determines if the object that shall be accessed is already cached locally.

```
1  class Data {
2      int value;
3  }
4
5  class Access {
6      Data obj = new Data();
7
8      void work() {
9          obj.value = 23;
10         return;
11     }
12 }
```

(a) Method `work()` with access to an remote object.

```
1 %g24820 = param('g', 'g', this+0+0, signed, no-helper)
2 access_check [%g24820],read,object,complete-object,check-id:0
3 %g24847 = (%g24820,'g').Access.obj
4 %i24870 = 23
5 access_check [%g24847],write,object,complete-object,check-id:1
```

(b) Method `work()` in LASM representation.

Figure 3.2: Translation from Java code to LASM with access check.

- A call to a function of the Jackal runtime–system which will transfer a copy of the object to the local node, if the object is nit available on the local node.

With these access checks the Jackal runtime–system is able to provide a S–DSM environment to the program.

The compiler backends implement several language independent standard code optimizations techniques such as common–subexpression elimination, invariant code motion, loop unrolling and the like. A full list of all supported optimizations can be found in [26]. In addition, the Jackal compiler also supports machine specific optimizations and special language specific optimizations. As Java is the programming language that was chosen for the implementation used in this thesis we will have now a closer look at the Java specific optimizations.

```
1 class A {
2     int[] array = new int[42];
3 }
```

(a) Class with a separate array object.

```
1 class A {
2     int[42] array;
3 }
```

(b) Class with inlined array object.

Figure 3.3: Optimization: Object inlining

Closed–World Optimization

Java's polymorphism and the feature to dynamically load classes prevent most of the optimizations, such as method–inlining and object–inlining. These optimizations require knowledge about the complete set of Java classes that are part of the application that is compiled, in order to produce good results. Therefore Jackal can compile applications with the so–called *closed–world assumption* that is an assertion the compiler that all classes are already known at compile time and that no code will be dynamically loaded at runtime. Programs that are compiled with *closed–world assumption* allow the compiler to use a large set of optimizations and using these more aggressively.

Inlining Optimizations

The programming model of Java application often leads to a high number of relatively small objects that references each other. By the *object inlining* optimization [8] the performance can potentionally be improved because the overhead of object creation, pointer dereferencing and garbage collecting can be reduced. Figure 3.1 shows the exemplary object inlining of an array. Note that the optimization shown can not be done by the programmer because Java does not provide the syntactical constructs that would be necessary. Objects are always inlined along with their header information, as for example the *vtable* of an object.

The high number of relatively simple methods that result of the Java programming model lead to a huge overheads incurring when a function is called. *Method inlining* is a well–known optimization technique to avoid these overheads. As requirend by the

semantics of object–oriented program the methods of its most–specific class have to be invoked for each object, eaven after casting an object to a less specific class. This prevents efficient method inlining. Although Jackal is able to perform efficient method inlining on proper final, static and private declared methods (see Figure 3.1).

Escape Analysis

Java semantics require that all objects that are created by the keyword *new* are allocated on the heap. This is also true for objects that are only used inside of a method. These objects use heap memory and therefore lead to *garbage collector* runs. Instead of allocating the memory for these objects on the heap Jackal is able to allocate memory for objects on the stack using *escape analysis* [4, 7]. By allocating the memory for those objects on the stack instead of the heap the actual object creation and garbage collection can be avoided, because the object is automatically deallocated when the function is exited.

Boundary Checks

Array accesses have to be guarded by bounds checks that ensure the correct memory usage. This can lead to *IndexOutOfBounds* runtime exceptions [5]. As array accesses occur quite often these checks have an innegligible influence to the performance of a program. Beside the unsafe option to completely disable these bounds checks Jackal can safely eliminate these checks that are known to repeat already successfully passed checks [28]. To archive this, Jackal performs *data–flow analysis* to track all array base pointers and related sets of index identifiers for which bounds checks already took place [31].

3.2 The Jackal Runtime System

The Jackal runtime system provides the almost complete Java API to the programmer. By the transformations that were performed by the Jackal compiler and the access checks that were inserted into to the intermediate code representation (see section 3.1) the runtime system is able to provide a S–DSM environment to the program.
The startup of a program that was compiled with Jackal's S–DSM support consists of three phases. At first the runtime system initializes the communication and networking subsystem on each node of the cluster. After the networking subsystem is prepared all the nodes establish their communication links to all the other nodes leading to an intermeshed communication network. After all the network connections have been established the node that was specified as the master node calls the main function of the

```
1  class B {
2      int counter = 0;
3
4      final void inc() { counter++; }
5
6      private void dec() { counter--; }
7
8      void work() {
9          inc();
10         ...
11         dec();
12         return;
13     }
14 }
```

(a) Class with separate methods.

```
1  class B {
2      int counter = 0;
3
4      final void inc() { counter++; }
5
6      private void dec() { counter--; }
7
8      void work() {
9          counter++;
10         ...
11         counter--;
12         return;
13     }
14 }
```

(b) Class with inlined methods.

Figure 3.4: Optimization: Method inlining.

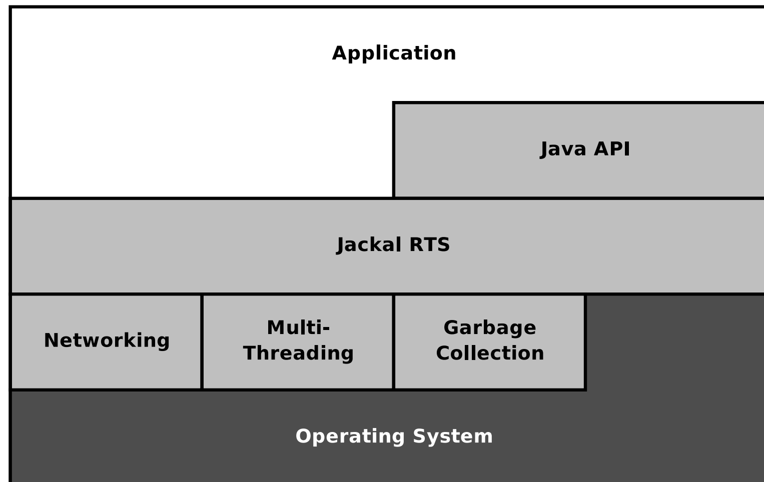


Figure 3.5: Architectural view of the Jackal runtime system.

program.

3.3 Architecture

The Jackal runtime system implements various level of abstraction to an application (see Figure 3.5).

The lowest level of abstraction is provided by the operating system of the nodes running a Jackal program. Operating system services like I/O, TCP/IP and memory allocation are supplied by system–calls of the kernel.

On top of the operating system are Jackals own networking library (Lizard) and multi–threading library (Taco). Using the Lizard library modules Jackal is able to provide DSM services on a wide range of networks—currently TCP/IP, MPI and InfiniBand are supported. The network library provides a stream–oriented interface for Jackals runtime system. Using the Taco library module Jackal is able to hide the various native thread libraries, such as the POSIX Threads, and provides a consistent thread interface.

The runtime system of Jackal resides on top of the low–level threading–module, the networking–module, and the operating system. The task of the runtime system is to supply the Java API and the user program with all features that are needed for the creation of a global address space on top of the cluster nodes. In addition to the cre-

ation, caching and transfer of objects between different nodes in the cluster using the DSM Protocol (see section 3.3.2), the Jackal runtime system must also perform *Garbage Collection.*

When a node tries to allocate memory for a new object and there is not enough memory left on that node the runtime systems initiates a local garbage collection to free memory of objects that are not used anymore. If the local garbage collection can not free sufficient memory in the object heap, the runtime tries to allocate the object on a remote node. In case that the runtime system can not find a node with enough free memory a global garbage collection run is initiated on all nodes in the cluster. If still not enough memory on a node could be free a *OutOfMemoryError* is thrown.

3.3.1 Memory Model

The Java memory model (JMM) [13] specifies that each thread has its own *working memory* that can be seen as a thread–local cache. The *main memory* of the program is used for communication between threads. Objects that are modified by a thread are flushed into *main memory* whenever a synchronization point is reached. These synchronization points are mapped to the beginning and the end of a synchronized block. The Jackal runtime–system conceptually copies all objects stored at the *working memory* of a particular thread to the *main memory* and invalidates these objects in the *working memory* of all other threads at the time a lock operation is reached. The first access of a thread to one of these objects will therefore copy it from *main memory* to the *working memory* of that thread anew.

Jackal stores all objects in a single, virtual address space that is shared among all threads. To achieve this Jackal introduces an uniform address–mapping among all nodes in the cluster. Memory is addressed by a tupel (node, 0x...) consisting of the rank of the home node of an object and the memory address of the object on that node. These Global DSM Object References (GOR) are created at the time of the execution of a new statement. A GOR does not change as long the object is not removed by the garbage collector. Therefore, it is an unique identifier for each living object.

The processor that executes a new statement allocates the memory for the object in its *object heap* and registers the GOR of that object at the runtime system. Because that node provides memory for the object and has registered its GOR it is called the *home node* of that object. If a node wants to access an object that is not locally available it has to perform two steps: First it must allocate physical memory in its *caching heap* in which the object can be stored. Second it must fetch an up to date copy of that object from its *home node* (see Figure 3.6). The local node then maintains a mapping of the

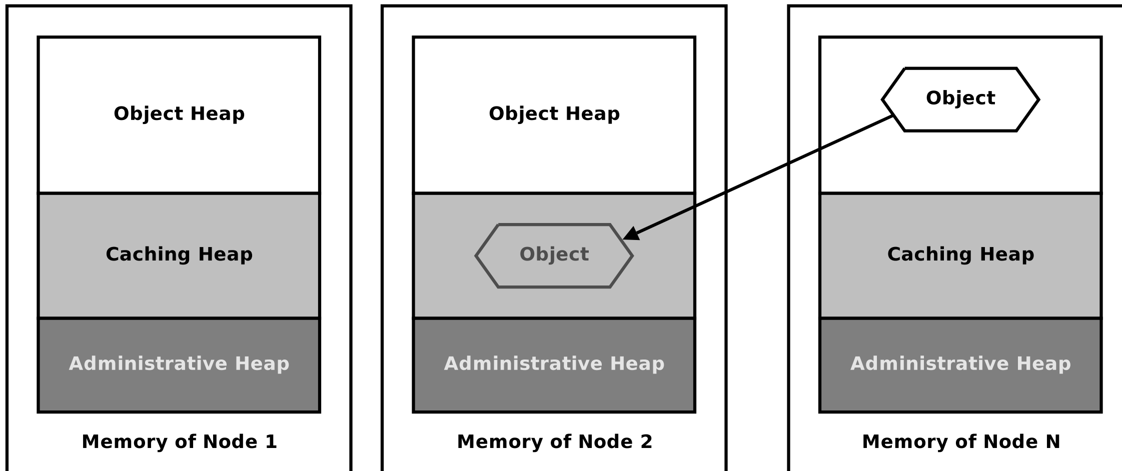


Figure 3.6: High–level view on the DSM provided by the Jackal runtime system.

local address of the object (in the *caching heap*) to the GOR of the object.

DSM Memory Layout

The heap memory on each node of the cluster is divided in three sections [26], as can be seen in Figure 3.6. The *object heap* is used to allocate memory on that node for objects that are created during the execution of the program. Access to objects that are not in the local *object heap* of the node, trigger a transfer from home node of the object to the local node. The runtime system will therefore request a copy of the remote object and then transfer this copy from its home node to the local node. Copies of such remote objects are placed by the runtime system in the so–called *caching heap*.

The last section of the heap memory is the *administrative heap*. This section of the heap is used by the runtime system to store data that is needed by its own to manage object accessibility's. To manage accessibility so–called *read–write bitmaps* are stored for each object. These bitmaps have to be maintained for all objects and arrays in each thread separately. If a node holds a read–only copy of an object only the read–bit in the bitmap for that object has to be set. Arrays which are also objects in Java are treated in a different way. To reduce false sharing, arrays are partitioned into contiguous but independent regions each 64 KB in size.

3.3.2 DSM Optimizations

The runtime system of Jackal is able to perform optimizations additional to those that are performed by the compiler (see section 3.1). In the following section the most important DSM optimizations will be discussed.

Access Check Elimination

The compiler can not determine all unnecessary access checks and therefore can not remove them. But most of these can be removed at runtime by the runtime system. This is, for example, possible for all objects on their home nodes, as these objects will always be present on these nodes. After the runtime system has migrated (see 3.3.2) an object from one node to another, all access checks on the new home node of the object can be removed, but therefore access checks on the previous home node must be inserted again.

To remove unneccessary access checks the compiler backend uses forward interprocedural data–flow analysis over the control–flow graph (CFG) of the program. An access check for an object o at a point p can be removed if the access check for o has already been checked on every path through the CFG that reaches p, and no synchronisation point lays on these paths [27].

Multi–Writer–Protocol

Jackal implements a so–called multiple–writer cache–coherence protocol that combines features of HLRC [32] and TreadMarks [16, 15]. As in the HLRC protocol, modifications to an object are only flushed to the object's home node. For this purpose each node that has requested a write able copy of an object stores a shadow copy of that object, the so–called shadow object. If the runtime system has to flush all changes that were made to an object back to the home node of that object, it calculates the difference the object with its shadow copy and then only transfers the difference between the two objects if any back to the home node of the object. This approach significantly reduces the amount of communication and allows concurrent writes to the same object.

Read–Only Replication

If all threads accesses a set of objects read–only the runtime system can create replica of these objects. The threads then receive these read–only copy's of the objects and no further remote accesses are necessary. Threads that hold read–only *object replications* are no longer forced to flush these replications to the objects home nodes.

Migration Strategies

Another optimization that can be performed by the runtime system is a technique called *home–node migration*. If there is only one node in the cluster that continuously requires write access to an object, this object is migrated from its initial home node to that node. This reduces the need to flush changes that have been made by the node to the former home node of the object.

For applications whose load–balance changes over time, or for non–dedicated clusters were the load on a node can change anytime by other programs *thread migration* [29] is required. Thread migration is the process of moving a thread from one machine to another to improve load balancing or to reduce the amount of necessary network communication by moving the working thread to the required data instead of transferring all the necessary data to the thread. Jackal supports thread migration even in heterogeneous clusters, allowing a thread to move between different architectures, for example from a IA32 machine to a IA64 machine.

3.4 Summary

In this chapter the basic features of the Jackal compiler and its runtime system along with the S–DSM protocol were described. Together they provide a fast and easy to use approach distribute a thread–parallel program.

The compiler translates the source code of a program and prepares the code during the transformations and optimizations for the S–DSM Jackal runtime system, by adding access checks before each object access. The runtime system then assures that each object that is accessed on a node is locally available. If the object was not created on the accessing node it is transferred by the RTS to the caching heap of the node. This creates a DSM transparently for the program.

4 Implementation and Optimization

This chapter will explain the data structures (see section 4.1) that were used to implement the LBM for two–dimensional and three–dimensional domains fluid dynamic simulations. After a in–depth–explanation of the data structure was given, the LBM algorithm itself will be analyzed. To increase the performance of the LBM algorithm various optimization techniques were used to on the sequential version of the algorithm. To further increase the performance the LBM was then parallelized and distributed on a cluster nodes. To improve the performance of the parallized version two additional optimization techniques were used.

4.1 Data Structures

During one time step of the LBM all cells in the domain are updated. In order to update one cell at time step t the discrete velocity distributions of the cell itself and all their neighboring cells of time step $(t-1)$ are needed.

In order to avoid data dependencies in the update algorithm, two distinct grids are allocated. One of the grids is the *src-grid* the other is the *dst-grid*. Therefore during one update iteration the distinct velocity values are only read from the src–grid and the result is written to the dst–grid. After the update iteration is completed the dst–grid holds the new velocity distributions for the next time step $(t+1)$. The dst–grid and the src–grid must be swapped after each time step in order to provide the correct velocity values for the next time step. By the introduction of two distinct grids all data dependencies in the update iterations can be removed.

Besides the values of the velocities per cell also the information whether a cell is a fluid cell or an obstacle cell must be stored. Instead of storing this information distinct in each cell another array that only stores this information is allocated. Therefore no encapsulation class for the cell information is needed.

To be able to change the memory layout of the data structures without the need to change the implementation of the algorithm, the grid data cannot be accessed directly but must be accessed through methods that are provided by the `Grid2D` or `Grid3D` classes. Figure 4.1 shows the interface as it is provided by the Grid2D class. These

```
1 public interface Grid2D {
2     double getValue(int x, int y, int i);
3
4     boolean isObstacle(int x, int y);
5
6     void setObstacle(int x, int y, boolean value);
7
8     void setValue(int x, int y, int i, double value);
9 }
```

Figure 4.1: Interface of the Grid2D class.

abstract grid classes not only encapsulate the data structures that store the velocity values but also maintain the information about the type of each cell.

Two different memory models were implemented for this thesis. In the first data model the single cells are part of an multi–dimensional array and are accessed via several indirect array accesses (see section 4.1.1 and 4.1.2). In the other data model the velocity values of the cells are stored in a vectorized way in a one–dimensional array. In order to access these vectorized cell elements index calculation must be performed by the encapsulating grid class (see section 4.1.1 and 4.1.2).

4.1.1 Two–dimensional Data Model

For the 2D lattice Boltzmann equations the *Q2D9* lattice model was chosen (see section 2.5.1). Therefore each cell in the domain consists out of 9 discrete velocity values that must be stored. This is implemented in a 9 element long array of double–precision floating–point numbers. To avoid problems with the array indexes, constants have been defined according to the discrete velocities of the Q2D9 model (see section 2.5.1).

2D Array Data Structure

As can be seen in Figure 4.2 the data layout for the `Grid2DArray` class is build of an array (y–direction) out of references to another array (x–direction) of references to the cell data array. Each of the cell data array consists of 9 double–precision floating–point numbers representing the 9 discrete velocities. The information about the cell type is stored in a separate two dimensional boolean array.

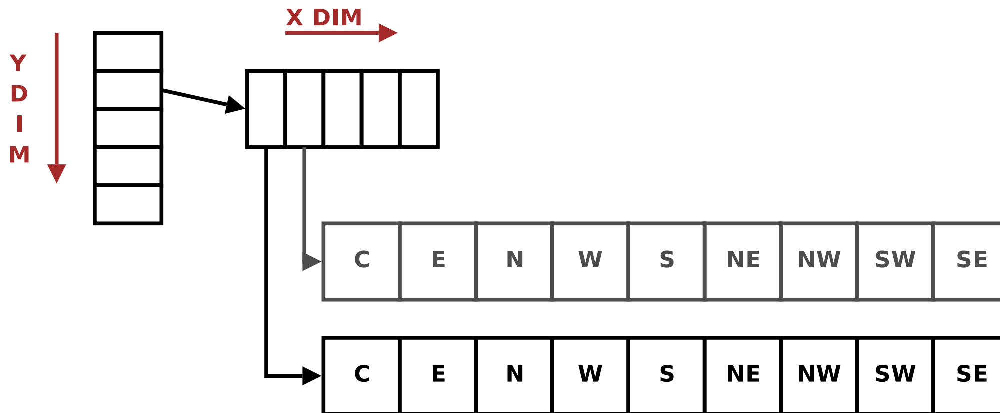


Figure 4.2: Memory layout of the 2D array data structure.

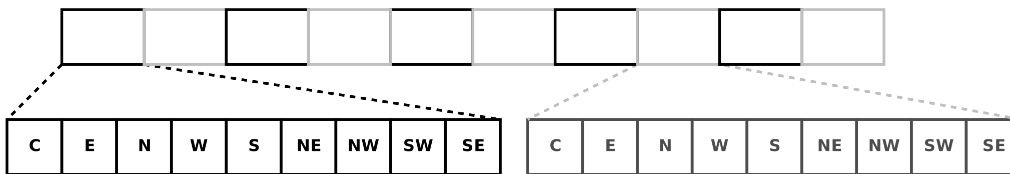


Figure 4.3: Memory Layout of the 2D vector data structure.

2D Vector Data Structure

The data structure for the `Grid2DVector` as can be seen in Figure 4.3 is build of a single array that has a length of $dim_y \cdot dim_x \cdot 9$ double–precision floating–point numbers. Where dim_y denotes the size of the domain in the y–direction and dim_x in the x–direction. In order to access a particular velocity value of a cell the following index calculation must take place in the `Grid2dVector` class:

$$index = (y \cdot dim_x + x) \cdot 9 + i$$

where x and y denote the coordinate of the cell in the domain and i denotes the discrete velocity value of the cell that should be accessed. The type information for each cell is stored in one–dimensional array of boolean values in a same way as the velocity values.

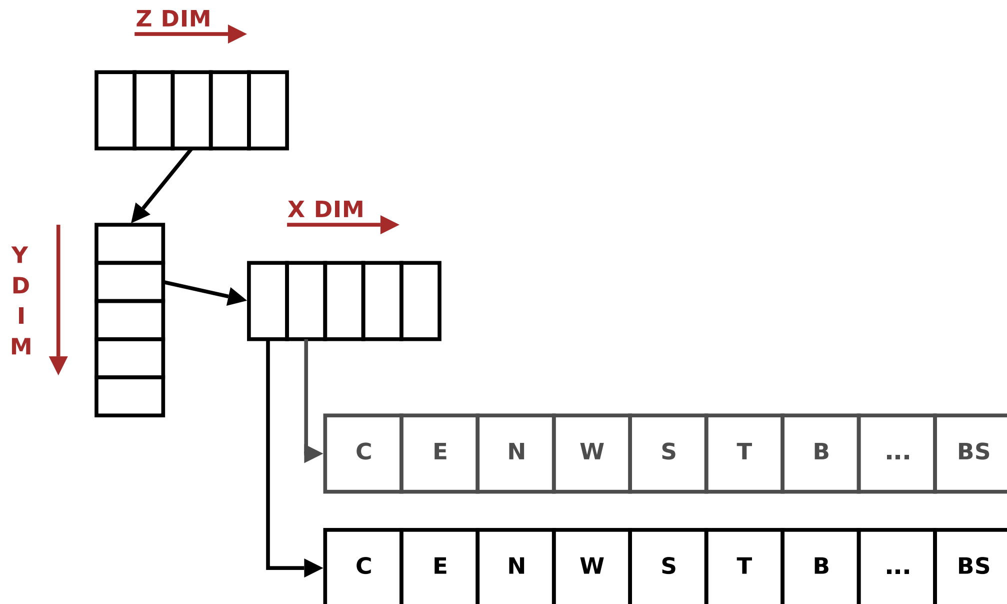


Figure 4.4: Memory Layout of the 3D array data structure.

4.1.2 Three–dimensional Data Model

The *Q3D19* lattice model was chosen for the 3D lattice Boltzmann equation (see section 2.5.2). Therefore each cell in the domain consists out of 19 discrete velocity values that must be stored. To avoid problems with the array indexes, in addition to the constants of the *Q2D9* model ten additional constants have been defined according to the additional discrete velocities of the Q3D19 model (see section 2.5.2).

3D Array Data Structure

The `Gird3DArray` class is build out of three time indirect arrays (see Figure 4.4), similar to the data layout for two–dimensional domains (see section 4.1.1). The first array stores the references in z–direction to a data structure that is equal to the data structure described in section 4.1.1. The type information in a 3D domain is stored in a similar way as in the 2D case. This information is stored in a three dimensional boolean array.

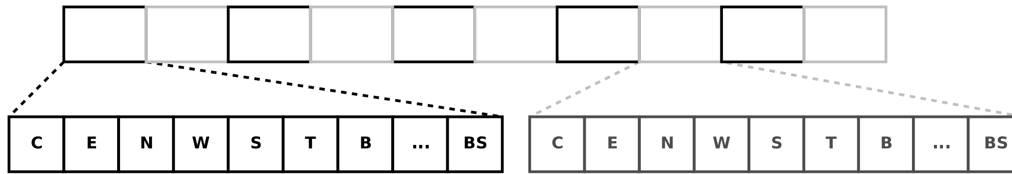


Figure 4.5: Memory Layout of the 3D vector data structure.

3D Vector Data Structure

As can be seen in Figure 4.5 the data layout in the `Grid3DVector` class is build out of single array with is capable of storing $dim_z \cdot dim_y \cdot dim_x \cdot 19$ double–precision floating–point numbers. In order to access a velocity value of a special cell index calculation must be done:

$$index = ((z \cdot dim_y + y) \cdot dim_x + x) \cdot 19 + i$$

where x, y and z denote the coordinates of a particular cell in the domain and i denotes a velocity value of the cell. Whether a cell is a fluid or a boundary cell is stored in a separate one–dimensional boolean array in the same way as the velocity values are stored.

4.2 Sequential Program Optimizations

In the following section all major optimization techniques will be described that were used to improve the performance of the LBM core. In general heavy use of `finalize` has been made in order to allow the compiler and the runtime system to optimize more aggressively [23]. The starting point of all optimizations is a relatively compact LBM core. Figure 4.6 shows the LBM core for the 2D solver:

The loops in line 1 and 2 are used to iterate over the whole domain that must be simulated. In line 3 to 6 the velocity values from the sourounding cells are copied to a local temporary array (this corresonds to the stream step, see Figure 2.1). In the loop from line 10 to 17 the velocity values in the current cell and the values from the sourounding cells are weighted and a new velocity distribution for the next time step is calculated (this corresonds to the collide step, see Figure 2.2).

```
1  for (int y = yFirst; y <= yLast; y++) {
2      for (int x = xFirst; x <= xLast; x++) {
3          for (int i = 0; i < Grid2D.CELL_SIZE; i++) {
4              tmp[i] = src.getValue(x - Grid2D.EX[i],
5                                    y - Grid2D.EY[i], i);
6          }
7
8          ...
9
10         for (int i = 0; i < Grid2D.CELL_SIZE; i++) {
11             u = Grid2D.EX[i] * ux + Grid2D.EY[i] * uy;
12             f = Grid2D.w[i] * (1.0 + 3.0 * u + 4.5 *
13                                u * u - 1.5 * u2);
14             dst.setValue(x, y, i, Constants.AGEMO *
15                          tmp[i] + Constants.OMEGA *
16                          rho * f);
17         }
18     }
19 }
```

Figure 4.6: An unoptimized 2D LBM core.

```
1  for (int y = yFirst; y <= yLast; y++) {
2      for (int x = xFirst; x <= xLast; x++) {
3          for (int i = 0; i < Grid2D.CELL_SIZE; i++) {
4              tmp[i] = src[y - Grid2D.EY[i]]
5                             [x - Grid2D.EX[i]][i];
6          }
7
8          ...
9
10         for (int i = 0; i < Grid2D.CELL_SIZE; i++) {
11             u = Grid2D.EX[i] * ux + Grid2D.EY[i] * uy;
12             f = Grid2D.w[i] * (1.0 + 3.0 * u + 4.5 *
13                                  u * u - 1.5 * u2);
14             dst[y][x][i] = Constants.AGEMO * tmp[i] +
15                            Constants.OMEGA * rho * f;
16         }
17     }
18 }
```

Figure 4.7: 2D LBM core with inlined data structure accesses.

Inlining

After a correct version of a LBM Solver for 2D (`LBM2DWorker`) and 3D (`LBM3DWorker`) had been implemented profiling these programs showed that most of the time was spent in the `getValue` and `setValue` methods of the encapsulating Grid classes.

Therefore the the Grid classes were extended in a way that they return a reference to their internal data structures. Because of that specialized solvers were written which are able to directly work on the array (see Figure 4.7) or vector data structures and reduce the overhead of calls to the `getValue` and `setValue` by inlining these methods [2].

Unrolling

After the overhead of the encapsulated data structures had been eliminated, further investigations of the generated machine code showed that the innermost loop over the elements of a cell had not been automatically unrolled by the compiler. Furthermore the calculation of the neighbor cells by the discrete velocity vectors (`Grid2D.EX[]`, `Grid2D.EY[]`) had also not been eliminated by the compiler.

To reduce the number of index calculations that are needed to access the correct neighbor cell these indexes were calculated manually. Thus the innermost loop over all velocity values of a cell was also removed by unrolling this loop. By unrolling the number of branches is reduced, leading to fewer branch–prediction errors and therefore to fewer pipeline stalls [9].

Blocking

Another possible way of improving performance is to use data locality and therefore reuse data that has already been loaded into the cache, this technique is called blocking [9]. If a element is accessed not only that element is loaded into the cache, but also a whole cache line is loaded. To use these elements the domain must not be processed line–by–line but must be divided in blocks (see Figure 4.9). Blocking reuses the elements that have already been loaded into the cache and therefore produces a higher spartial locality, but also introduces a higher overhead for each of the blocked loops. Whether the higher spartial locality results in a better overall performance or the higher overhead for each blocked loop dominates must be decided from case to case.

4.3 DSM–specific Optimizations

After the sequential version of the program had been optimized the LBM solver was thread parallelized. This was done in a way that each thread is given the starting and the end index for each dimension of the domain. The size of the subdomains, that are calculated by seperate threads, can be configured at runtime. Through the thread parallelism the Jackal S–DSM system is able to distribute the program over all cluster nodes.

This section will describe optimizations that were introduced to adjust the LBM solvers to the special needs of a software–based DSM system as described in section 3.2.

4.3.1 Thread–parallel Memory–allocation

In the serial program version the memory was allocated in one pice before the actual LBM solver was started. On a DSM system the initial memory allocation is done by the master–program thread on the root node. As now the various LBM solver threads are started on the remote cluster nodes they start processing their parts of the domain. The data structures that were allocated by the master–program thread on the root node must now first be transferred from the *object heap* of the master node to the *caching*

```
1  for (int y = yFirst; y <= yLast; y++) {
2      for (int x = xFirst; x <= xLast; x++) {
3
4      tmpC = src[y][x][C];
5      tmpE = src[y][x - 1][E];
6      tmpN = src[y - 1][x][N];
7      tmpW = src[y][x + 1][W];
8      tmpS = src[y + 1][x][S];
9      tmpNE = src[y - 1][x - 1][NE];
10     tmpNW = src[y - 1][x + 1][NW];
11     tmpSW = src[y + 1][x + 1][SW];
12     tmpSE = src[y + 1][x - 1][SE];
13
14     ...
15
16     u = ux;
17     dst[y][x][E] = Constants.AGEMO * tmpE + factor
18             * (1.0 + 3.0 * u + 4.5 * u * u - u2);
19
20     u = uy;
21     dst[y][x][N] = Constants.AGEMO * tmpN + factor
22             * (1.0 + 3.0 * u + 4.5 * u * u - u2);
23
24     u = -ux;
25     dst[y][x][W] = Constants.AGEMO * tmpW + factor
26             * (1.0 + 3.0 * u + 4.5 * u * u - u2);
27
28     u = -uy;
29     dst[y][x][S] = Constants.AGEMO * tmpS + factor
30             * (1.0 + 3.0 * u + 4.5 * u * u - u2);
31
32     ...
33
34     }
35 }
```

Figure 4.8: 2D LBM core with unrolled innermost loop.

```
1  for (int yy = yFirst; yy <= yLast; yy += BS) {
2      yStart = (yFirst > yy) ? yFirst : yy;
3      yEnd = ((yy + BS) < yLast) ? (yy + BS) : yLast;
4
5      for (int xx = xFirst; xx <= xLast; xx += BS) {
6          xStart = (xFirst > xx) ? xFirst : xx;
7          xEnd = ((xx + BS) < xLast) ? (xx + BS) : xLast;
8
9          for (int y = yStart; y <= yEnd; y++) {
10             for (int x = xStart; x <= xEnd; x++) {
11
12                 ...
13
14             }
15         }
16     }
17 }
```

Figure 4.9: 2D LBM core with blocking.

heap of these nodes (see section 3.3.1).

As described in section 3.3.1 the grain of distribution are objects or huge arrays chunks of 64 KB. As each cell of the domain is an array that only stores 9 doubles (in the 2D model) or 19 doubles (in the 3D model) each cell will be transferred in a separate communication packet by the runtime system. This leads to a enormous amount of packages that must be send from the root node to each of the other nodes in the cluster. The following example will show the network load that is introduced by the central memory allocation:

For a 2D model with 1000 cells in x–direction and 1000 cells y–direction with 4 distinct cluster nodes, the domain is divided into 4 subdomains of equal size. Each thread has to work on 250000 cells. Therefore 750000 cells must be migrated from the master node to the 3 remote nodes during the calculation of the LBM solvers on these nodes.

To reduce the traffic thread–parallel memory allocation was implemented. Therefore the main thread on the master node only allocates the 2 dimensional array of references to the single cells. The memory for the single cells is then allocated on the single nodes that processes the corresponding subdomain. This dramatically reduces the amount of

message transfer that must be exchanged between the cluster nodes.

As for the described 2D domain with 1000 $\times$ 1000 cells this reduces the amount of messages from about 75000 to 4000 per iteration.

4.3.2 Association of Boundary Cells

After the communication overhead at the program startup has been eliminated through the thread–parallel memory–allocation (see section 4.3.1) only the neighboring cells between nodes must be exchanged between the nodes after each time step. As the level of distribution is an object or an array with at most 64 KB (see section 3.3.1) each cell is transferred in a separate packet. This leads to a huge amount of packages.

In a 2D domain with with 1000 cells in x–direction and 1000 cells y–direction and 2 distinct cluster nodes, after each time step each node must fetch 1000 cells from the other node. Therefore 2000 messages must be transmitted between the nodes in each time step.

In the MPI programming model the programmer would transfer the 1000 cells together to the other node leading to 2 messages that must be transferred per time step between the nodes of the cluster.

To reduce the amount of messages that must be transferred after each time step the jackal system would have to analyze access patterns to remote objects. As the access to the remote cells does not change over time the runtime system should be able to associate cells automatically and transfer them in a single message.

At the moment this can not be done automatically but must be done by the programmer. Therefore a special method `associate(Object o1, Object o2)` is provided by the Jackal runtime system. With this method the programmer can associate object o_2 to the object o_1. This association tells the runtime system that every time object o_1 has to be transferred to another node object o_2 shall be transferred, too. The runtime system can transfer the two objects in one single message.

Figure 4.10 for example shows a 2D domain that was split in 9 subdomains. Each of these subdomains is processed by another thread on different cluster nodes. Neighboring cells in a subdomain that have the same color have been associated with each other and will be transferred in one single message. As described in Chapter 2 in order to update a cell velocity values from the sourounding cells are needed. Therefore thread *5* needs to get all the boundary cells of its northern, eastern, southern and western neighbouring subdomains. Thread *7* for example also needs the boundary cells of subdomain *8* for the update of its cells. The cells that have to be transfered from subdomain *8* to the

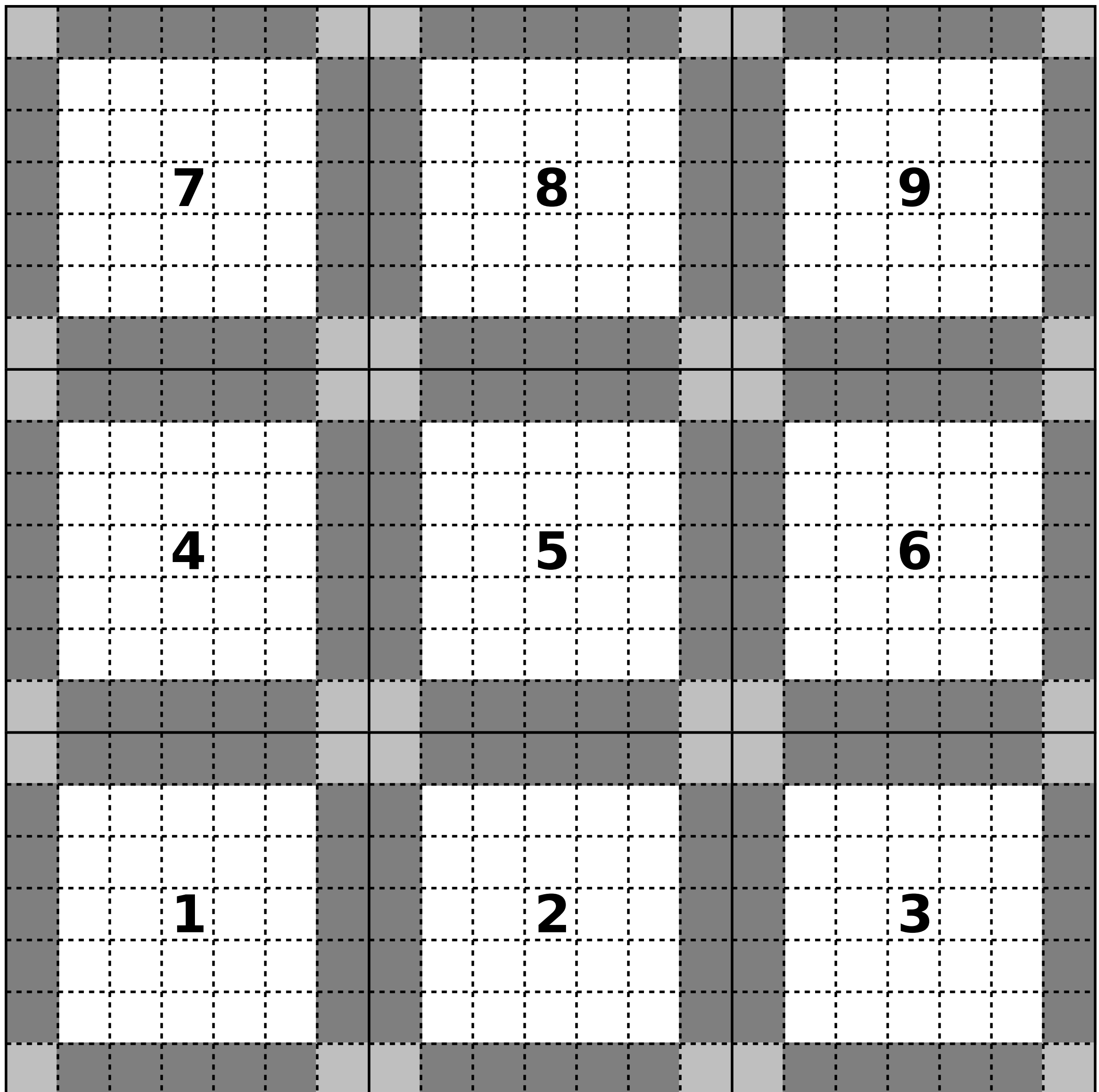


Figure 4.10: 2D Domain: with 9 subdomains.

subdomain *5* and *7* do overlap in the south–weterns cornercell. Because of this overlap the corner cells must not be associated with any other cell to avoid superfluous data transfers.

Regardless of the separate transfer of the corner cells the association reduces the amount of messages that must be transferred between the two nodes in the example from 2000 messages per time step to 6 messages.

4.4 Summary

This section described in detail the data structures that were used to implement the lattice Boltzmann method. In addition, the various limitations of different data layouts were explained.

It was shown how various optimization techniques such as inlining, unrolling and blocking were used to enhance the overall performance of the LBM core. Afterwards the solvers have been thread parallelized so that they could be distributed by the Jackal S–DSM system on number of distinct nodes. The solvers have then been adopted to the special needs of a software–based DSM to gain reasonable performance.

5 Performance Evaluation

In this chapter we will evaluate the performance gain achieved by applying the various optimization techniques of section 4 to the sequential LBM core as well as for the S–DSM parallelized LBM core. We evaluated several different Java compiler (see Table 5.2) and measured the performance that was achieved for the 2D and 3D LBM problem domain. Out test setup consisted of four identical computing nodes. The machine characteristics are listed in Table 5.1:

5.1 Two–dimensional Testcase

As a 2D testcase a problem domain with 1000×1000 cells and 50 iterations was chosen. 50 iterations over a 1000×1000 domain result in total of a 50 millions of lattice updates per simulation run.

Single Node Performance

Although the workstations in the cluster have four CPUs each the following measurements were only performed with a single thread running on one CPU in order to get the performance and Mega Lattice Updates per second (MLU/s) per CPU.

As can be seen in Table 5.3 and Figure 5.1 the inlining and unrolling optimizations increase the performance of the LBM core. We can further observe that the *Vector* memory model results in a better performance then the *Array* memory model in all cases.

Component	Type
CPU	AMD Opteron (x86_64), 4 CPUs, 2.2 GHz
L2 cache	1 MB
Total memory	16 GB (DDR 333)
Networking	1 GBit/sec
Operating System	SUSE LINUX 9.2 (x86_64), kernel version 2.6.5-7.191-smp

Table 5.1: Test setup data.

Compiler	Version	Optimization options
JDK	1.4.2_08-b03	–
GCJ	4.0.0	-O3 -fno-bounds-check -ffast-math
Jackal	build-2044	-no_bounds

Table 5.2: Compiler setup data.

Compiler	Memory Model	Simple	Inline	Unroll	Blocking
JDK	**Array**	45.5	31.8	15.9	15.1
	Vector	44.6	31.2	13.7	14.4
GCJ	**Array**	34.0	15.3	12.9	12.9
	Vector	27.6	11.8	7.0	7.5
Jackal	**Array**	29.3	24.0	13.0	13.1
	Vector	24.7	18.8	8.5	9.2
Jackal DSM	**Array**	78.6	55.1	28.8	30.2
	Vector	49.2	33.7	10.4	11.5

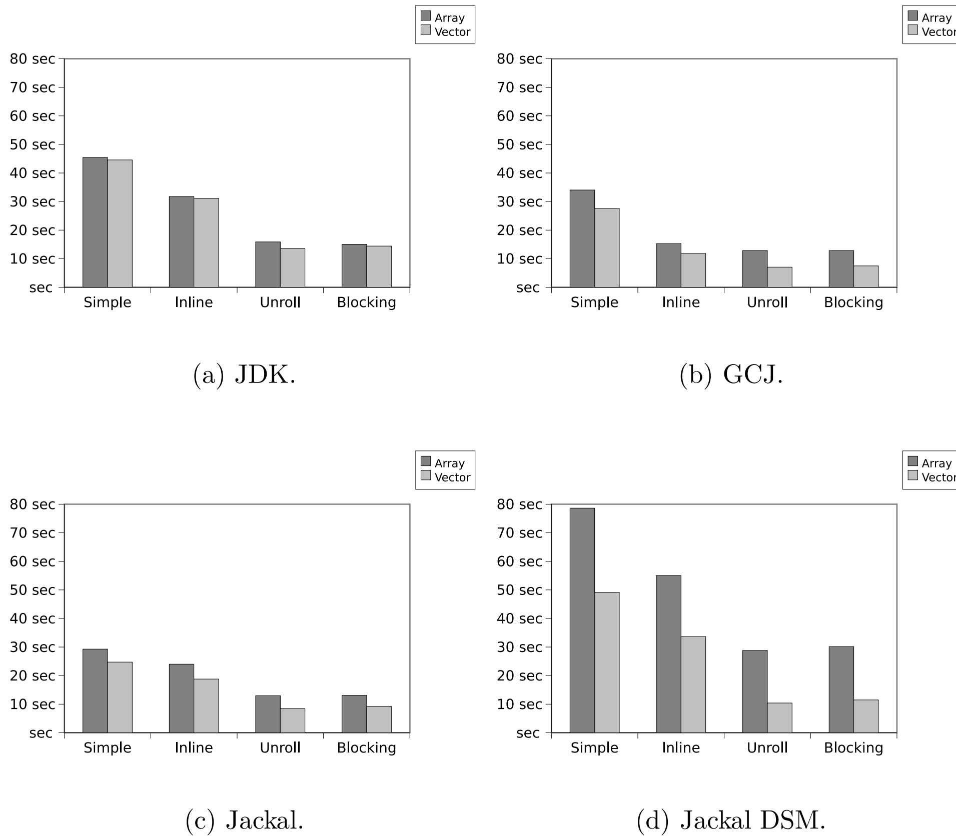
Table 5.3: Runtimes of the benchmarks for the 2D case with different compilers.

For the 2D case the overhead for blocking the loops outweighs the higher spacial locality.

Table 5.4 shows the Mega Lattice Update per second (MLU/s) for various optimizations and different compilers. The highest update rate is achieved for each compiler by the unrolled version of the LBM core. Best results are achieved by the GCJ compiler followed by the Jackal compiler without DSM support. Due to the huge overhead introduced by the access checks of the DSM-enabled Jackal compiler the performance of the single–node DSM–enabled LBM roughly equals the performance of the JDK compiler.

Compiler	Simple	Inline	Unroll	Blocking
JDK	1.1	1.6	3.7	3.5
GCJ	1.8	4.2	7.1	6.7
Jackal	2.0	2.7	5.9	5.4
Jackal (DSM)	1.0	1.5	3.7	3.5

Table 5.4: MLU/s of the benchmarks for the 2D case with different compilers.



(a) JDK. (b) GCJ.

(c) Jackal. (d) Jackal DSM.

Figure 5.1: Runtimes of the benchmarks for the 2D case with different compilers.

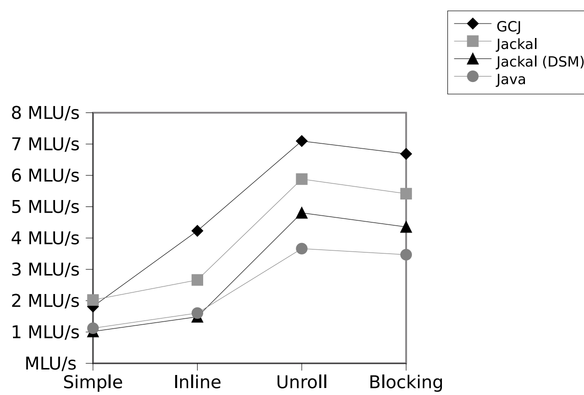


Figure 5.2: MLU/s of the benchmarks for the 3D case with different compilers.

# of nodes	normal			thread–local allocation			association		
	t	speed–up	MLU/s	t	speed–up	MLU/s	t	speed–up	MLU/s
1	38.2	1.0	1.3	31.1	1.0	1.6	30.2	1.0	1.7
2	23.2	1.6	2.2	19.8	1.6	2.5	14.7	2.1	3.4
4	19.3	2.1	2.6	16.4	1.9	3.1	12.5	2.4	4.0

Table 5.5: Runtimes of the DSM benchmarks for the 2D case with different optimizations.

DSM Performance

For the DSM testcases, up to four nodes in the cluster have been used. Although the node have four CPUs each the following measurements were only performed with a single thread running on one CPU per node. This was done to get the performance and MLU/s rate per CPU by providing the CPU with the maximum bandwidth of the networking interface.

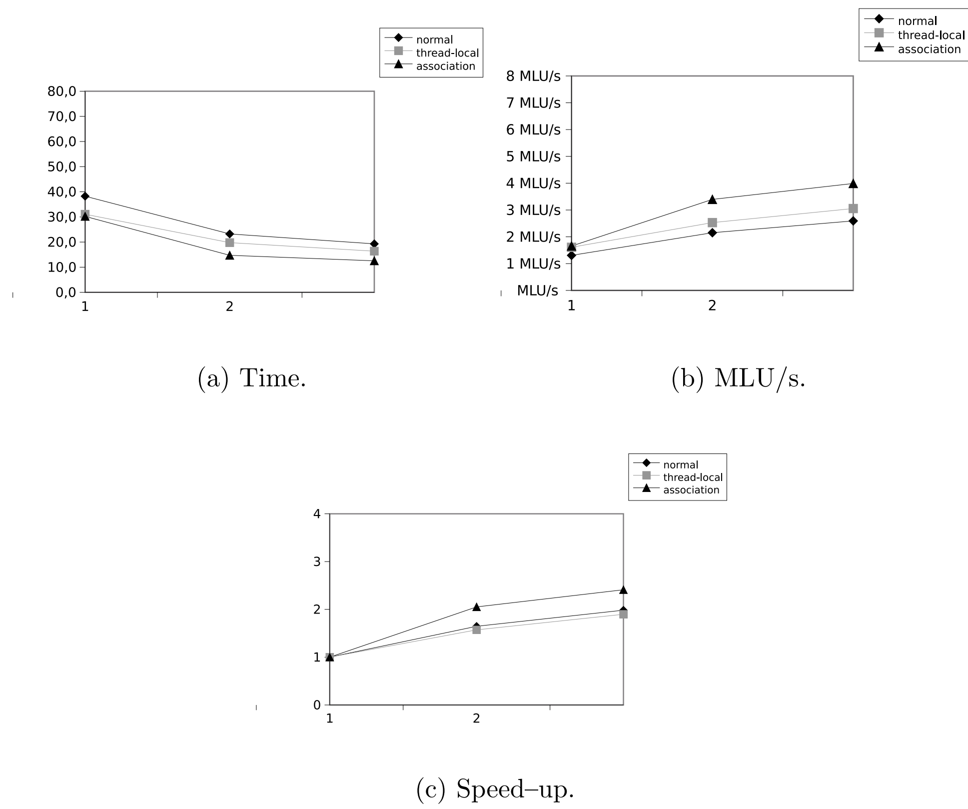
Table 5.5 shows the results of the fastest sequential LBM core (*Unroll*) on up to four cluster nodes with the various DSM–specific optimizations applied. As one can see, the dominant factor for the performance of the LBM core in a S–DSM system is the communication between the nodes. The high impact of communication in due to the high latency of the underlying network. Thread–local allocation of data is able to reduce the number of messages sent, because each of the threads stores the date needed for computation in its local heap. Thus communication only occurs whenever a boundary cell is accessed. Another gain of about 34% in performance was achieved by associaton of the boundary cells.

5.2 Three–dimensional Testcase

As 3D testcase a problem domain with $100 \times 100 \times 100$ cells and 50 iterations was chosen. As for the 2D testcase, the 3D testcase also needs a total of 50 millions of lattice updates per simulation run .

Single Node Performance

As for the 2D the following measurement were performed with a single thread running on one CPU in order to get the performance and MLU/s rate per CPU.



(a) Time.

(b) MLU/s.

(c) Speed–up.

Figure 5.3: Runtimes of the DSM benchmarks for the 2D case with different optimizations.

Compiler	Memory Model	Simple	Inline	Unroll	Blocking
JDK	**Array**	125.6	94.0	39.5	40.5
	Vector	125.9	102.0	35.1	36.5
GCJ	**Array**	106.6	56.2	44.5	35.2
	Vector	89.2	48.3	23.7	23.5
Jackal	**Array**	88.9	68.6	30.7	29.8
	Vector	90.6	66.1	21.7	19.0
Jackal DSM	**Array**	175.4	126.7	49.0	46.1
	Vector	116.2	90.4	38.1	33.5

Table 5.6: Runtimes of the benchmarks for the 3D case with different compilers.

Compiler	Simple	Inline	Unroll	Blocking
JDK	0.4	0.5	1.4	1.4
GCJ	0.6	1.0	2.1	2.1
Jackal	0.6	0.8	2.3	2.6
Jackal (DSM)	0.4	0.5	1.4	1.4

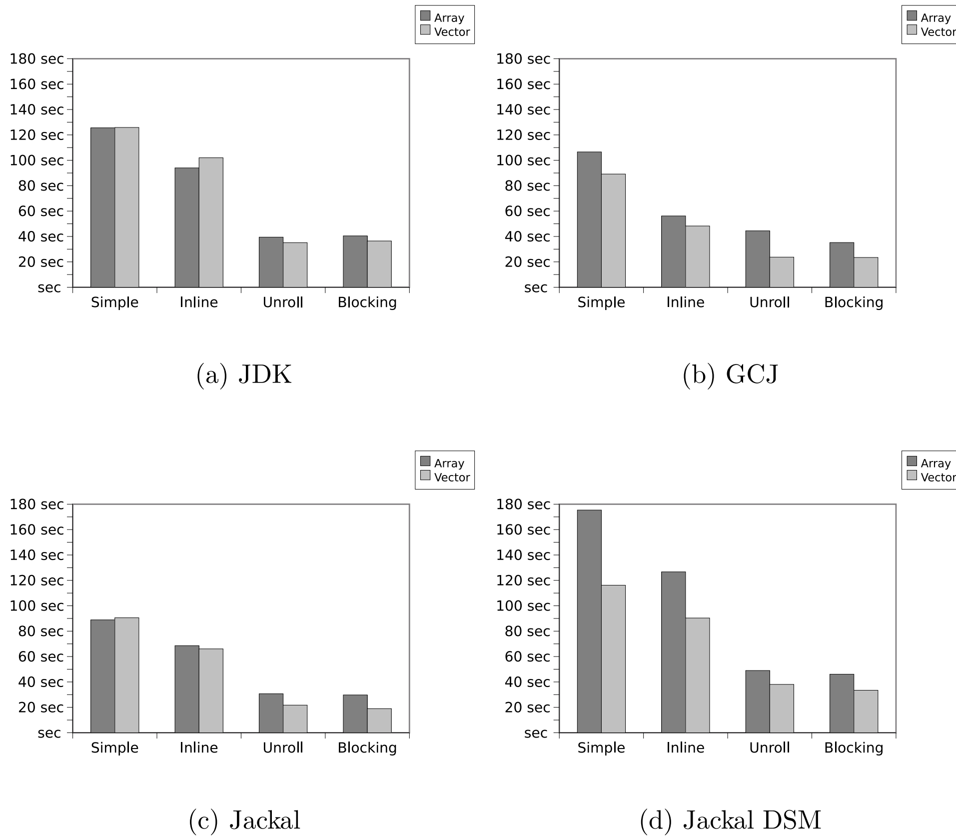
Table 5.7: MLU/s of the benchmarks for the 3D case with different compilers.

As can be seen in Table 5.6 and Figure 5.4 inlining and unrolling optimizations increase the performance of the LBM core by maximal 74%. But unlike in the 2D case in the 3D case blocking further increases the performance of the LBM cores that were compiled by the GCJ and Jackal compilers. Because of the higher number of floating point operations that are necessary in order to perform a single 3D lattice update the higher spacial locality outweighs the overhead for blocking the loops. As in the 2D testcase the *Vector* memory model results in a better performance then the *Array* model in all cases by about 25%.

In Table 5.7 the MLU/s rates for the various optimizations and compilers can be seen. For each compiler the highest update rate is achieved by the blocked version on the LBM core. Best results are achieved by the Jackal compiler without DSM support enabled. As in the 2D testcase the DSM–enabled Jackal compiler performance is roughly comparable to the JDK compiler.

DSM Performance

As for the 2D testcase, up to four nodes in the cluster have been used. For the following measurements only one thread running on one of the node's CPUs was used. Thus, the



(a) JDK

(b) GCJ

(c) Jackal

(d) Jackal DSM

Figure 5.4: Runtimes of the benchmarks for the 3D case with different compilers.

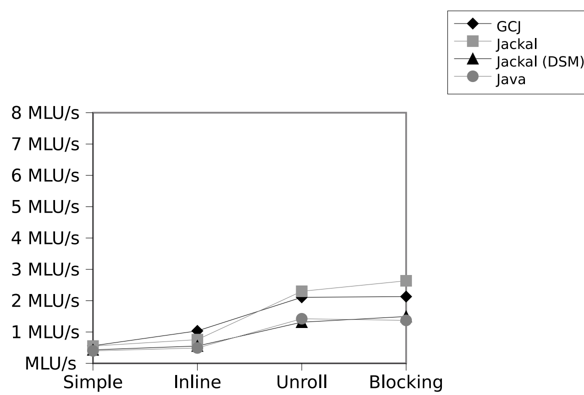


Figure 5.5: MLU/s of the benchmarks for the 3D case with different compilers.

# of nodes	normal			thread–local allocation			association		
	t	speed–up	MLU/s	t	speed–up	MLU/s	t	speed–up	MLU/s
1	89.8	1.0	0.6	77.2	1.0	0.6	67.4	1.0	0.9
2	83.2	1.2	0.6	71.0	1.2	0.7	60.5	1.6	1.4
4	57.6	1.3	0.7	36.3	1.6	0.8	21.4	2.7	2.3

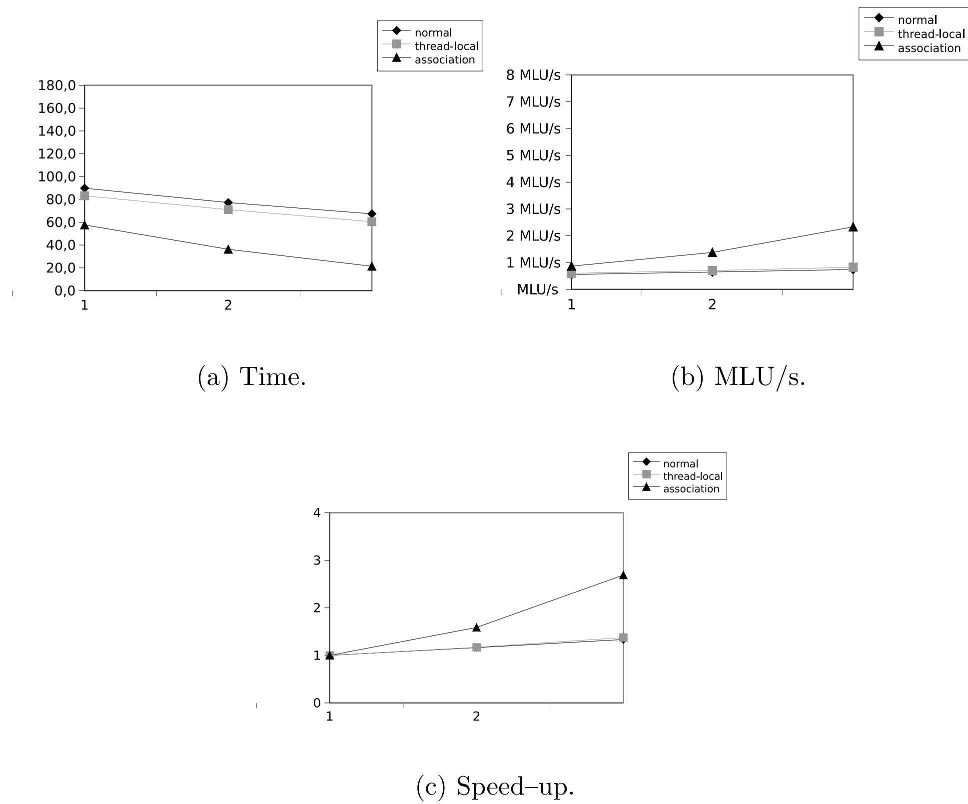
Table 5.8: Runtimes of the DSM benchmarks for the 3D case with different optimizations.

CPU was provided with the the maximum bandwidth of the network interface.

Table 5.8 and Figure 5.6 show the result of the unrolled LBM core on up to four nodes with various DSM–specific optimizations applied. Although the unrolled LBM core is not the fastest it was chosen in order to be able to compare the results with the 2D unrolled LBM core. One can see (as in the 2D case) that the dominant factor for the performance of the LBM core in a S–DSM system is again the communication between the individual nodes. Therefore the same S–DSM specific optimizations such as thread–local allocation and associations was used to gain a higher performance boost.

5.3 Summary

In this chapter an overview over the various two–dimensional and three–dimensional testcase was given. We showed how the performance in the sequential 2D and 3D LBM core could be increased by a factor of 3. Further more we demonstrated how the latency in the S–DSM enabled LBM cores could be reduced by specific optimizations such as *thread–parallel allocation* and *association* by about 60%.



(a) Time.

(b) MLU/s.

(c) Speed–up.

Figure 5.6: Runtimes of the DSM benchmarks for the 3D case with different optimizations.

6 Related and Future Work

This chapter gives an overview of related work in the field of parallelization and S–DSM systems. Beside this also an overview to possible extensions of the Jackal S–DSM system will be given.

6.1 Related Work

One way to distribute and parallelize programs is by means of a message–passing libraries such as the MPI [12, 11]. These libraries hide the details of the communication from the programmer, but communication between the logical nodes in the cluster must be explicitly added to the code. Each communication has to be written explicitly by using special communication primitives. These allow sending and receiving of data messages and using special auxiliary functions such as parallel I/O for example.

Another way of for parallelizing is the use of Remote Procedure Calls (RPC) [19, 18] or Remote Method Invocation (RMI) [25]. In contrast to MPI these two methods abstract from *data shipping* to *function shipping*. With function shipping we denote that one object sends a invocation request to another object on a remote node in the cluster. In the case of RPC these remote functions must specified by the programmer. For the Java specific RMI classes that shall be capable of offering remote methods must implement the `Remote` interface. For classes that implement the remote interface the Java RMI compiler is able to generate a special *stub* class and *skeleton* classes which then handle communication between remote and local objects.

Besides these traditional distribution and parallelization approaches S–DSM systems as Jackal [26] and TreadMarks [16, 15] simulate a global address space on all cluster nodes to hide memory bounds of the single nodes. Data distributing is done automatically in a transparent way. Whenever data is referenced that is not locally available on one node it is automatically transferred from its remote node to the local node. Jackal is a fine–grain S–DSM system that does communication on the level of objects and array chunks. TreadMarks instead uses a page–based system to identify memory regions that must be transferred from one node to another.

6.2 Future Work

As already described in section 4.3 thread–local allocation was used to reduce the amount of communication. This allocation is only possible for the multi–dimensional data layout. But as performance evaluations on a shared memory system have shown, the vectorized data layout achieve a better performance. The Jackal system should be able to also provide support the vectorized data layout on a S–DSM system. By automatically splitting and distributing vectors among all cluster nodes such that the memory for each part of the vector is allocated on the node that will work on it.

Another optimization that was implemented to further reduce the amount of network communication is to associate the boundary cells between the various subdomains. This was done in a way that the access to the first element of such a set of boundary cells triggered the transfer of all boundary cells from that node in a single message. The Jackal runtime system should be able to analyze the working set of a thread. This enables the runtime system to identify a set of remote objects that are transferred from a remote node to the local thread. The runtime system could then automatically associate these objects and therefore reduce the amount of messages between the nodes.

In numerical simulations special status information changes over time and therefore have to be transferred after each time step. Typically this information gets only updated by a single thread. Then, this status information must be transferred from the master thread to either all other nodes in the cluster or to special groups of nodes. To further decrease the amount of messages that have to be send, Jackal should use *broadcast* packages to send messages to all nodes in the cluster or *multicast* to only send a message to a special group of cluster nodes. As more and more routers and switches are capable of transmitting broadcast and multicast package this could reduce the latency.

7 Summary

In this thesis a 2D and 3D fluid dynamic simulation based on the lattice Boltzmann method was implemented. The LBM represents an alternative approach to the traditional solvers that are based on the traditionally Navier–Stokes solvers. It is based on cellular automata, where the whole domain if build of distinct cells. All cells holds a finite number of discrete states and get updated synchronously at each discrete time step.

To parallelize the LBM implementation the Jackal framework was chosen. The Jackal framework consists of a compiler and a runtime system. Together these two components provide a shared memory on the distributed physical memory of the cluster. The Jackal compiler is a ahead–of–time multi–pass compiler that transparently inserts access checks before of each object access. These access checks are basis of the DSM implementation.

In this thesis various optimization techniques were used to improve the performance of the sequential LBM core. After a reasonable speed–up was achieved the LBM solver was thread parallelized. By the thread parallelism the Jackal S–DSM system is able to distribute the program over all cluster nodes. To reduce network load at first thread–local domain allocation was implemented. Therefore, the domain must not be transferred from the home node to the other nodes. Each nodes allocates the memory for its subdomain locally. To further optimize network load boundary cells have been associated with each other. The association of cells enables the runtime system to transfer all cells together.

Our performance evaluation shows that by optimizing the lattice Boltzmann method the sequential program we are able to achieve 5.5 millions of lattice updates per second for a 1000×1000 grid. By further optimizing the sequential program in a distributed shared memory specific way we are able to achieve a speed–up of 3.4 in both 2D and 3D test case. This results in a total of 2 millions of lattice updates per second.

A Commandline Options

This appendix will give an overview over all supported command line options that can be used to configure the program.

[--associate] Associate the boundary cells between the several nodes (default = false). Associated cells are transferred together in a single network packet. NOTE: Implies parallel switch (see parallel for more details).

-d <dimension> Dimension of the problem [2 | 3].

[--it <iter>] Number of time steps that will be simulated (default = 0).

[-m <memory>] Use one of the following memory layouts: [Vector | Array] (default = Array).

[--nx <nx>] Number of threads in x-direction that will be used (default = 1).

[--ny <ny>] Number of threads in y-direction that will be used (default = 1).

[--nz <nz>] Number of threads in z-direction that will be used (default = 1).

[-o <output>] Name of the file where to write the OpenDX output to (default = result.dx) or "NULL" (no output is written).

[--parallel] Use the parallel memory allocation (default = false). NOTE: Currently ONLY the Array model is supported.

[-t <type>] Use one of the following solver types: [Simple |

```
                Inline | Unroll | Blocking ] (default = Unroll).

[-x <xDim>]     Number of cells in x-direction (default = 4).

[-y <yDim>]     Number of cells in y-direction (default = 4).

[-z <zDim>]     Number of cells in z-direction (default = 4).

[-h|--help]     Print this help text.
```

Bibliography

[1] G. Antoniu, L. Bouge, P. J. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded java bytecode for distributed execution (distinguished paper). In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, pages 1039–1052, London, UK, 2000. Springer-Verlag.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[3] P. Bhatnager, E. P. Gross, and M. K. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94(3):511–525, 1954.

[4] B. Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 20–34, New York, NY, USA, 1999. ACM Press.

[5] R. Bodik, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[6] M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[7] J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

[8] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming language design and implementation*, pages 7–17, New York, NY, USA, 1997. ACM Press.

[9] K. Dowd and C. Serverance. *High Performance Computing.* O'Reilly, 1998.

[10] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software Practice and Experience*, 30(3):199–232, 2000.

[11] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Massachusetts/USA, 2000.

[14] M. Griebel, T. Domseifer, and T. Neun. *Numerical simulation in fluid dynamics.* SIAM monographs on mathematical modeling and computation. SIAM, 1998.

[15] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, volume 29, pages 213–226, 1995.

[16] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, San Francisco, California/USA, 1994.

[17] R. Mei, W. Shyy, and D. Yu. Lattice boltzmann method for 3d flows with curved boundary. Technical Report NASAgCR-2002-211657, ICASE, NASA Langley Research Center, Hampton, Virginia, June 2002.

[18] Sun Microsystems. XDR: External Data Representation standard. RFC 1014, June 1987.

[19] Sun Microsystems. RPC: Remote Procedure Call Protocol specification. RFC 1050, April 1988.

[20] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, New York, New York/USA, 1996.

[21] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, and D. A. Wood. Cost-effective fine-grain distributed shared memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, 1998.

[22] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, California/USA, 1994.

[23] R. Simmons. *Hardcore Java*. O'Reilly, Cambrige, Massachusetts/USA, 2004.

[24] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, Great Britain, 2001.

[25] CA Sun Microsystems Inc., Mountain View. *Java Remote Method Invocation*, 1997.

[26] R. Veldema. *Compiler and Runtime Optimizations*. PhD thesis, Vrije Universiteit, Amsterdam, the Netherlands, October 2003.

[27] R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, C.J.H. Jacobs, and H.E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–92, Snowbird, Utah/USA, 2001.

[28] R. Veldema, T. Kielmann, and H. E. Bal. Optimizing java specific overheads, java at the speed of c ? In *Proceedings of the European High Performance Computing and Networking*, pages 52–60, Amsterdam, the Netherlands, 2001.

[29] R. Veldema and M. Philippsen. Near overhead-free heterogeneous thread-migration (accepted). In *Proceedings of Cluster2005*, Boston, Massachusetts/USA, 2005.

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

[31] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. R. Altman. A java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 128–138, Newport Beach, California/USA, 1999.

[32] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 75–88, Seattle, Washington/USA, 1996.