FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Concepts of waLBerla Prototype 0.0

J. Götz, S. Donath, Ch. Feichtinger, K. Iglberger, U. Rüde

Technical Report 07-4

Concepts of waLBerla Prototype 0.0

J. Götz, S. Donath, Ch. Feichtinger, K. Iglberger, U. Rüde
Lehrstuhl für Systemsimulation
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany
{goetz, donath, feichtinger, iglberger, ruede}@informatik.uni-erlangen.de

March 27, 2007

In recent years the lattice Boltzmann method has achieved the reputation as a true alternative to the classical Navier-Stokes approach to simulate fluid behavior. Due to this development and the multitude of different Lattice Boltzmann solvers developed at our chair for different problems, the desire for a single Lattice Boltzmann implementation grew. This paper introduces the waLBerla project, an all-in-one Lattice Boltzmann solver for a wide range of applications. Next to the basic requirements of the easy adaptivity and extensibility for new fluid problems, the waLBerla project also aims at physical correctness, high performance and ease of use. In this paper, we will describe the first working Prototype 0.0 focused on creating a common base to build on in the next development stages.

1 Introduction and Motivation

The lattice Boltzman method (LBM) enables fast and flexible fluid simulation, and is easily extended for physical applications. The aim of the waLBerla project is to combine several years of experience in a wide range of applications in one code. waLBerla is an acronym for widely applicable Lattice Boltzmann algorithm from Erlangen. Besides supporting existing applications ranging from moving particles over free surfaces and charged colloids to blood flow, the code is intended to feature a foundation for easy creation of new extensions. waLBerla provides well-defined interfaces and a suitable framework supporting parallelization, I/O and visualization mechanisms. Thus, from a programmer's point of view, waLBerla is a comprehensive tool library, while from a user's perspective, it is a powerful stand-alone simulation program.

Since it is going to be a large long-term project, it is splitted in several small successive parts. In this report, besides the goals planned for the whole waLBerla project, technical details of the first prototpye in version 0.0 and concepts for coding and environment are documented. However, the following sections explain the principles only briefly. More technical details are available in [FGID07].

2 Goals of waLBerla

waLBerla is planned to be a comprehensive program rich in features as well as a library for easy development of new applications based on fluid simulation. Thus, it is supposed to meet the requirements of scientific researchers, performance optimisers, code developers as well as needs for industrial production of simulation results. The currently planned final version of waLBerla will include a wide range of physical applications already profoundly researched by our group:

- fluid in arbitrary geometries with complex boundary conditions like periodic, acceleration, freeslip, a.s.o., in order to be able to support medical applications like blood flow in vessels (cf. [Göt06])
- fluid–structure interaction to simulate moving particles in a fluid, including the calculation of the forces that occur between the particles and the fluid and therefore determine the movement of particles caused by the fluid and vice versa (cf. [Igl05])
- simulation of liquid fluids having free surfaces, which includes both bubbles and atmosphere, resulting in scenarios that enable simulation of squirting drops, rising and deforming bubbles, foams, a.s.o. (cf. [KPR+05, Thü07])
- ionised fluid reacting to electrical fields induced by charges or charged colloids in the fluid (cf. [Fei06])

Instead of solely supporting the single applications, we plan to realize every possible combination of the mentioned features, enabling new scenarios by merging existing know-how and using the synergy effects.

Besides the applications, for accuracy, stability and computability reasons sophisticated techniques will be required and supported: Since for accurate simulations often high resolutions would be needed, resulting in not satisfiable memory requirements, grid refinement techniques will be indispensable. For the same reason, parallelization will be implemented, as well as a grid compressing technique as introduced in [PKW+03].

The domain setup will be task of the waLBerla program completely transparent for the user, i.e. the user will specify only the setup of the scenario and waLBerla autonomously decides where to refine the grid and how to decompose and distribute the domain on the cluster nodes. Especially for moving particles and free surfaces (i.e. moving bubbles), these decisions have to be changed dynamically during runtime, which results in adaptive grid refinement and dynamic re-decomposition of the domain.
An idea to realize this is to subdivide the domain in so-called patches that are computed independently from each other and communicate locally in the same process, or globally via MPI. These patches will be implemented in different flavors such that highly optimized primitive patches and specialized ones for specific applications exist. As a result, a patch would have to be able to transform itself from one type into the other, according to the actual state of the simulation.

In addition to these features other more technical issues are planned in order to enhance the usability and correctness of the simulation software:

- Configuration File:
  The user describes the complete scenario setup by writing a single configuration file that holds all information.
- Geometry Data:
  For the geometry setup, different possibilities will exist. E.g. the user describes a simple geometry in the configuration file, or specifies a geometry file that contains a geometry description, for example a triangular mesh exported from a CAD software.
- Physical Correctness:
  At an initial phase of the simulation, the input data is to be examined for physical correctness. This way it is ensured that the parameters for simulation setup are valid.
- Visualization:
  Several possibilities for exporting the simulation results will be implemented. Full-data visualization with 3-D visualization software (e.g. ParaView, Tecplot, OpenDX) as well as statistical data for graph plotting (e.g. gnuplot, xmgrace) will be supported.

- Tracking of Simulation Progress:
  During simulation, observation of key quantities will be possible, in order to check whether the simulation remains within valid parameters (e.g. the maximum velocity, global density, coordinates of objects, a.s.o.)

- Debugging:
  A sophisticated debugging system will help for both code as well as model development. The debugging system will include logging messages of different verbosity levels, asserts, recording of data accesses as well as detailed monitoring of selected cells.

To ensure the reliability and integrity, an automatic validation test mechanism will be implemented, which verifies the correctness of single modules and performs validation of the simulation. These tests will be computed automatically, sampling the results and warning the developers when irregulrities are encountered. The regular recurrence of the tests ensures that a faulty modification of the code will be discovered immediately.
Since waLBerla is intended to be a flexible widely usable software, it will run platform independent and thus can be executed under different operating systems with different compilers and architecure sets.
As a secondary goal, performance optimization techniques will be used to the extend to which they may be applicable.

3 Prototype 0.0

The initial step of the waLBerla project is to plan and develop a first prototype which will be released with version 0.0. This prototype is a simple lattice Boltzmann solver capable to compute only primitive scenarios. As an application, a Lid Driven Cavity was implemented. While the kernel and application itself is kept simple, the first parts of the embedding framework were set up like they are planned for the final version. Thus, the development of a sophisticated application will be much easier.

The framework components implemented in this first prototype handle:

- parsing of a configuration file (see subsection 5.1)
- checking of physical correctness (see subsection 5.2)
- debugging including logging, monitoring data access and monitoring cells (see subsection 5.3, subsection 5.5 and subsection 5.6, respectively)
- output of data files for visualization by ParaView (see subsection 5.7)

Furthermore, different data layouts are implemented by a single class respecting the conclusions of [WZDH05] (see subsection 5.8) and different so-called "Sweeps" are realized (see subsection 5.4).

In this prototype the patch idea is not yet implemented. Since the Lid Driven Cavity does not need sophisticated requirements of special applications and thus no different treatment in different areas of the domain, the code structure is kept simple. The communication between patches will be the main task for the next version. In further prototypes, some parts of the Prototype 0.0 will be renamed and incorporated in constructs of higher hierarchical level.

4 Class Diagram

For the project planning of waLBerla, UML and UML-like diagrams were used. Figure 1 shows the classes that handle all necessary data components. The basic class `DataLayout` is used by the different

Field classes (PDF, Velocity, Density, Flags) to store and access the data (for implementation details see subsection 5.5). The Grid class combines the Fields to one data component which serves as interface for the solver.

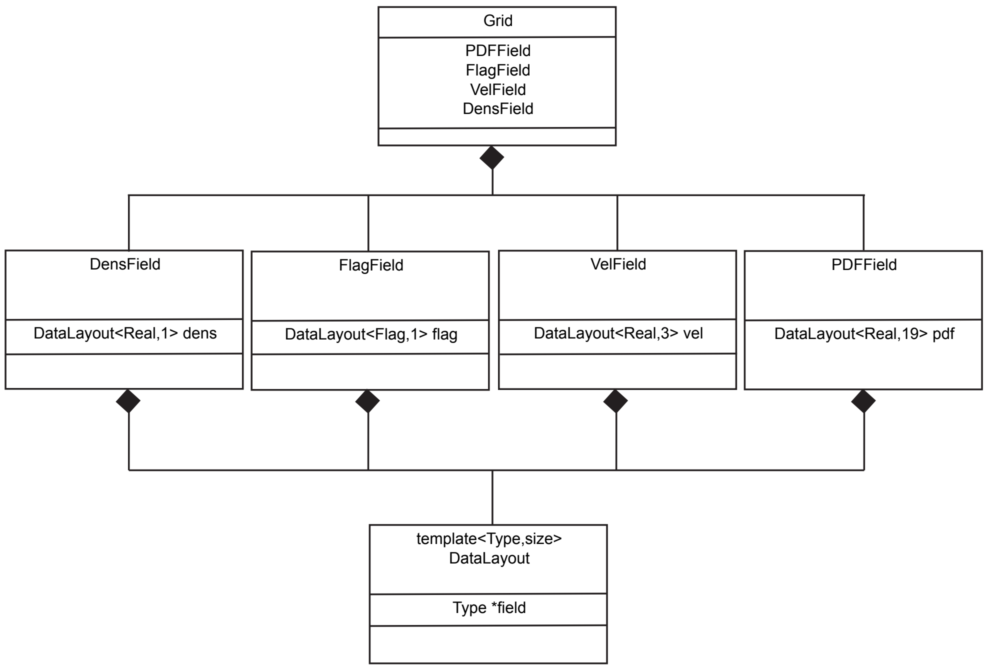

Figure 1: Class diagram of the data management classes

In each time step the Solver performs one sweep over the whole domain. The time step consists of BoundaryConditions, Stream, CalcMacro, Collide and if necessary additional application dependent functions. Figure 2 shows a dependency diagram of the respective modules. This part of the software is not implemented by hierarchical classes but in a C style, because otherwise code reproduction would occur when new higher-skilled patches (with more complicated Sweeps) are added.

5 Implementation Details

In this section we present selected details of implemented parts of waLBerla. During an extensive planning phase different possibilites for technical implementations were discarded before a decision was eventually reached. Alternative concepts which were not implemented are discussed in section 7.

5.1 FileReader

The FileReader class is the program's interface to the user which specifies a parameter file. The program once creates an instance and initiates the parsing of the file. Thereafter all data is stored in special data containers. In order to support future extensions of waLBerla and the information set specified in the parameter file, the FileReader is kept highly flexible. Making only syntactical restrictions, the data is not yet interpreted. The structure of data in the configuration file is organized in named blocks, that allow an arbitrary number of key–value pairs or hierarchically cascaded blocks. For convenience, also C-like

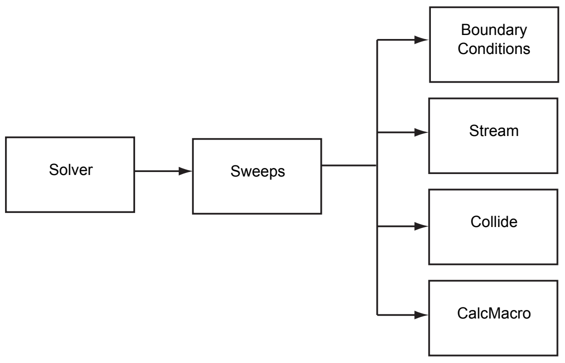

Figure 2: Dependency diagram of the solver

comments in the file are supported.
Classes that use the configuration data request specific blocks and `FileReader` returns the appropriate key–value pairs. Interpretation of the data occurs in the classes that process the data. Thus, these have to specify the name and the type of the information of interest. Therefore, `FileReader` reads the data without interpretation using the string datatype. When the data processing class requests a data it uses the `GetParameter` function which is able to convert the datatype appropriately to the specified type.

5.2 PhysicalCheck

The task of the `PhysicalCheck` class is to check the simulation parameters for coherence and validity. This class can be configured by the programmer according to the actual needs of the application: The header file contains definitions of mathematical formulas that describe all physical quantities (for all lattice Boltzmann applications) and their relationship to each other. In a second routine, the programmer defines which values (appearing in the specified formulas) are needed by the waLBerla program. Additionally, a check function has to be specified which checks the computed value for physical validity. For waLBerla, all physical and lattice values are computed and stored in a separate `SimData` struct.

The `PhysicalCheck` class provides much help to the user to set up a valid, physically correct scenario. For each quantity that is to be computed, the functions check, whether it can be computed by the set of quantities given in the parameter file. If it can be computed, the value will be checked for physical validity and stored. If the user has specified the same quantity in the configuration file, the result of the computation will be compared to the given value. In case of a mismatch, an error will be thrown. If it is possible to compute the sought quantity by several ways, all these ways will be computed and the results compared with each other. If the results differ, an error will be returned (although this should only happen when the set of formulas given is inconsistent). When the `PhysicalCheck` class detects that the sought quantity is not computable with the given set of quantities, it returns a string that contains information about which quantities or combination of quantities have to be specified in the configuration file such that the value can be computed.

Depending on the logging and debug level (see subsection 5.3) chosen, the `PhysicalCheck` class gives differently detailed information about issues concerning the computability and computation of values. Furthermore, for debugging purposes, the programmer can convince `PhysicalCheck` to not interrupt the program in case of inconsistent specifications by setting an appropriate flag.

5.3 Logging

The `Logging` class is a single-instance class providing atomic logging functionality. One static instance ensures the correct order and immediate flushing of outputs into the logging file. The class provides three different log functions for logging errors, warnings and informations and thus represents a primitive tool for a code-global logging system.

The logging system of waLBerla supports 4 different logging (verbosity) levels. The verbosity of information messages is controlled by different preprocessor macros. Additionally, errors and warnings can be logged or surpressed. By using appropriate `LOG_*` preprocessor macros the log function calls are inserted or removed depending on the definition and value of the preprocessor macros `NWARNING` and `LOGGING`. Thus, when disabling logging, no function call has an impact on performance. The same mechanism is used for `get` function calls (see subsection 5.5), where a macro is replaced by a fast `Get` or a sophisticated checking `MonitoredGet` function depending on `NDEBUG` macro.

For the programmers, there is no specification for the use of the logging macros and the decision which verbosity level to choose for a log message. This way, the logging mechanism provides a flexible tool for debugging and recording while not being too ornate.

5.4 Sweeps

A “Sweep” is the traversal of the grid, performing different computations for each cell. Thus, a Sweep routine is called by the time loop and describes the tasks to be done in each time step. Different Sweep routines adapted to different applications are assembled by several “atomic” functions which are inlined. These functions are designed to perform a basic step on a cellular basis. Thus, e.g. for an application working on an empty channel, a Sweep could be assembled by a stream, a collide and some boundary handling function, while a free surface application would need a Sweep that includes additional computations concerning the fill values, surface tension and moving of interface cells. The assembling and the choice of the appropriate Sweep is currently by the programmer. In later releases the choice will be done automatically depending on the needs of the scenario described by the user.

This also enables optimization for performance. By a clever decomposition of the domain, waLBerla later can use differently skilled patches that are optimized for their application. E.g. in a canal with an obstacle in the middle, three patches can be used, where only the patch in the middle has to treat boundary effects inside the domain while the others can be optimized for pure fluid, only performing boundary treatment at the boundaries of their subdomain.

5.5 Accessing Data

Data access is encapsulated by the `Field` classes (e.g. the `PDFField`, cp. section 4) which feature appropriate `Get` and `Set` functions for every purpose. For each funtion two overloaded versions exist, offering the possibility to either access a data by its coordinates `x`, `y` and `z` or an index. Additionally, the classes contain helper functions to modify and organize the data.

For debugging purposes, every function that accesses or changes data has a “`Monitored`” counterpart (e.g. `MonitoredGet` or `MonitoredRemove`). By setting appropriate preprocessor definitions, the standard `Get` function call is replaced by its `MonitoredGet` complement. These `Monitored` functions perform extra data checks, e.g. for valid coordinates, correct physical values, or perform extra logging like monitoring a cell (see subsection 5.6).

5.6 Monitoring Cells

The cell monitoring feature enables the user to monitor all value changes in one or more specific lattice cells. When waLBerla is compiled in debug mode, the access to data occurs by the `Monitored` functions (see subsection 5.5). Then the user specifies the cells of which data access shall be logged in more detail by simply adding an appropriate section containing the coordinates of the cell to the parameter file. Then, every `Monitored` function gives detailed information about the state of this cell every time when it is called. Thus, it is possible to track wanted and unwanted changes and to detect software bugs that may only become manifest in incorrect data.

5.7 Visualization

waLBerla is intended to be a software, which can be used easily and without any additional costs. Thus, the same has to hold for the visualization of the results.
Since many visualization tools exist, the waLBerla framework will be linked to different common formats and applications. Despite that OpenDX is free, experience shows that it is not easy to use. A novice needs a lot of time to get aquainted with until he is able to visualize the data. Thus, we focus on the open-source software Paraview [Par07]. Paraview features a flexible and intuitive user interface and supports distributed computation models to process large data sets. It is available for Linux, Windows and Macintosh computers and thus backs the platform independency of waLBerla. The software handles the following data formats:

- Raw data files
- VTK (visualization tool kit [VTK07]) files
- Different proprietary input formats.

A flexible and open format is the VTK-format, which supports legacy, serial and XML based file formats. The serial formats are easy to read and write either by hand or programmatically. It is also possible to export time-dependent data in a time series. Here, a number is added to the filename to denote the time step. However, these formats are not flexible enough to handle distributed data sets. In a first step, a serial paraview writer was implemented in waLBerla. It supports ASCII and binary output, where ASCII is reasonable and practicable for small sets since the data can be viewed in an editor. Reading binary data in Paraview is much faster than the ASCII pendant, thus recommended for visualization. Since Paraview processes legacy binary files only in big endian format, the binary output is swapped on Intel architectures before writing to file.
In a second step the XML output will be supported, which then allows distributed data sets which is important for parallelization.

5.8 Data Layouts

The `DataLayout` class represents a container for arbitrary data that is organized by a multi-dimensional array. It features different array layouts by accessing the data in different index orders according to the conclusions of Wellein et al. in [WZDH05]. They pointed out that for IA32 based machines performance of a code based on a multi-dimensional data array can be improved when using a "structure-of-arrays" layout instead of an "array-of-structures". Since even further improvements are possible by other nontypical index-permutations, the `DataLayout` class is designed to support every possible combination of index orders.

Therefore, internally a one-dimensional array is allocated and addressed according to the chosen layout by using the offset formula:

$$\text{Index} = a * x + b * y + c * z + d * l\,,$$

where x, y, z and l are the variables specified by a `Get` function to access the data item l at the cell coordinates (x, y, z), and a, b, c, d are strides that are defined according to the chosen data layout. For an "array-of-structures" layout, this results in:

$$\begin{aligned} a &= \mathrm{NrItems} \\ b &= \mathrm{DimX} * \mathrm{NrItems} \\ c &= \mathrm{DimY} * \mathrm{DimX} * \mathrm{NrItems} \\ d &= 1 \end{aligned}$$

while a "structure-of-arrays" layout is realized by:

$$\begin{aligned} a &= 1 \\ b &= \mathrm{DimX} \\ c &= \mathrm{DimY} * \mathrm{DimX} \\ d &= \mathrm{DimZ} * \mathrm{DimY} * \mathrm{DimX} \end{aligned}$$

Thus, besides the shown "zyxl" and "lzyx" permutations, also other combinations are possible, e.g. the "zylx" layout which is proposed in [Don04] as being optimal for performance on a Pentium IV architecture.

Since all functions accessing the data are inlined, it is expected that the compiler is able optimize the overhead of additional index calculations compared to a standard implementation by the use of loop-invariant code motion.

6 Coding Concepts and Environment

For a long-term project with fluctuating number of team members it is inevitable to chalk up coding concepts and a uniform environment that ensures a development standard and code quality. Therefore, coding standards, documentation and mechanisms for ensuring validation as well as architecture independency were installed.

6.1 Coding Style and Quality

To enforce uniform structure and allow easy interpretation by others, we introduced a coding style that standardizes the notation of names of folders, files, classes, functions, members, variables and preprocessor macros. The team members are urged to use descriptive names for functions and variables. A user-defined type "Real" for floating point variables is introduced such that e.g. switching between `float` and `double` precision is easily possible. Furthermore, it is emphasized to adhere to the so-called "const correctness". A secondary goal is to minimize compiler warnings for the various compilers used.

6.2 Target Architectures

As outlined in section 2, waLBerla is designed to support as many different target platforms as possible. Thus, the programmers have to respect the differences of architectures and compilers. This is especially weightly for Linux and Windows based systems where system-dependent functions differ considerably. Therefore, the use of such functions has to be limited or, if it is inevitable, respective system-dependent solutions are required. An existing example for this is the `Timing` class that uses system functions that differ under Linux and Windows.

To ease the switching of different platforms, project compilation is based on *CMake* [CMa07] which is a cross-platform make system. Independent from the platform, *CMake* detects a suitable compiler, examines the system environment and generates Makefiles for Linux or Visual Studio project files for

Windows, respectively, according to the configuration files that are set up once for the whole project. Due to the automatic detection of the compiler, switching between different compilers on the same architecture is not easy, unfortunately.

6.3 Documentation

For easier familiarizing with the waLBerla project code for new developers, a detailed programmer's documentation is realized. Due to the extensive work load which would be introduced by a documentation created separately, we use an automatic documentation system called *doxygen* [Dox07]. This system interpretes specially marked comments in the source code and generates a Latex or HTML documentation. For a uniform appearance, a documentation style was defined that prescribes the dense of information as well as the place appearing in the code. This way, every function and member is documented in detail in the source code, resulting in a self-explaining code.

6.4 Validation

For a simulation software both validation and verification is important. Therefore, different scenarios will be set up and key quantities will be compared to corresponding numbers that can be found in literature.

For verification during development each class of waLBerla comes with an own test program that checks for correct functionality and ensures that code changes do not change the correctness of waLBerla. It is planned to realize these validation and verification tests by an automated system working on the code version management system, autonomously detecting and reporting failures.

6.5 Benchmarking

An automated benchmarking system regularly runs different scenarios on different platforms. The performance values are logged in a `CSV` file annotated by relevant facts describing the system and environment as well as the code revision of waLBerla. This benchmarking system ensures a detailed logging of changes in performance during the development of the code and enables comparisons of computer architectures or different implementations.

7 Other Concepts not Implemented

In the course of planning and development of Prototype 0.0 many ideas emerged and were discarded. This section outlines the most important alternative concepts and discusses the reasons why they were not implemented.

7.1 Patch Inheritance

In the first steps of the planning phase it was planned to realize differently skilled patches by inheriting them from a primitive patch class. Then, the primitive patch would have contained all necessary functions like traversing the grid, performing the stream and collide step as well as the handling of boundary conditions. A more sophisticated patch, e.g. a patch handling moving particles, could inherit all basic functionalities of the primitive patch, only changing the functions that differ. However, it is clear that each patch with a sophisticated application would have a different body in the time loop (the so-called "Sweep"). That means, the order of steps processed every time step would change and new steps would have to be added. Code duplication would occur, and the result is that several different sweeps would exist in the patches. As a second consequence, the problem of the derivation order would occur. If a free-surface patch is derived from a moving-particle patch, it supports both functionalities. In order to get a patch being capable to handle only free surfaces, a second patch derived from the primitive patch would have to be designed. This would lead to massive code duplication, such that changes made in one patch have to be replicated in the second one, too.

Consequently, the design was changed. The respective part of the software concept does not base on classes but is constructed in a C-like style. The decision was to split the functionality of a sweep in atomar steps and reassemble them according to the needs of the patches. Thus, there is a collection of different sweep functions that combine the collide step, streaming step, a.s.o. in an appropriate way to fulfill the requirements of any application. That means that every atomar step has to be designed universally such that it fits into every application using it. This way, the code reproduction is minimized to the loops that traverse the grid. However, this approach has disadvantages. Since a sweep consists of several small functions, compiler optimization capabilites are reduced and possibly an overhead is introduced. Thus, two different approaches will be tested: The first approach uses atomar functions acting on one cell, being called from inside the grid-traversing loop. In order to minimize the function call overhead, all atomar functions will be inlined. The second approach uses atomar functions which traverse the whole grid one time. This results in less function calls. However, the grid is traversed several times per time loop, which presumably results in lower performance due to the increased memory traffic.

7.2 Data Layouts

Initially, the `DataLayout` class was planned to provide different data layouts, transparently to the `Field` classes by featuring a uniform set of access functions for each data layout. As data layouts, both array-based and a list-based layout should be implemented. However, due to the inherent differences of list and array-based data structures, the interface would be very complicated. Moreover, the manner of traversing the “grid” differs drastically. Thus, a design proposition that includes mechanisms to support both layout types revealed to be very unflexible in use and combined the disadvantages of both layouts. A sweep over the whole grid would not be realized by three `for` loops as usual in an array-based code, which can be easily optimized by the compiler, but by stepping forward cell by cell, requesting the next cell from the data layout class until the end is reached. The functions providing such an interface that behaves like an iterator are not simple and a overhead would be introduced. Moreover, it would be inevitable to have an indirect adressing for either reading or writing access for both the list-based and the array-based layout. A further consequence is that in order to provide good code quality, the data layout would have to be chosen by template parameters. However, this template parameter would influence the whole class hierarchy, because every class using the data layout class would have to support this parameter, too.

Therefore, a design decision dictates that list-based layouts will not be used, and different array-based layouts are realized by an adaptive addressing scheme as described in subsection 5.8. The lost advantage of saving memory for porous geometries featured by the list-based data layout will hopefully be counterbalanced by the patch technique in combination with the compressed grid technique (introduced in [Göt06] and [Igl03], respectively).

7.3 Logging

For the logging system ideas for a more sophisticated system were emerged. This concept included the possibility to switch on or off the logging messages for each type, module and class separately by using appropriate preprocessor macros. It was planned to create a special header file containing the macros, where the single options can be grouped and organized. Thus, with other master macros larger parts of the logging system could be switched. Discussions revealed that such a system is not only for the programmers very complicated to implement, but also the user will loose control very soon. Thus, only a global verbosity level was introduced and the programmer is in duty to decide which logging messages belong to which verbosity level.

8 Conclusion

The waLBerla project is a comprehensive software suite combining all LBM-related knowledge of our group in one single program. It is efficient, extensible and designed for a wide range of applications, including optimizations in terms of computability and performance. When completed, the waLBerla software will be a tool that features usability, easy parametrization, grid refinement, visualization, parallelization, debugging, benchmarking as well as validation mechanisms. It is an all-in-one solution for production use, LBM software development and experimental investigation of new physical applications. This report outlines the functionality implemented during the first project phase, the development of Prototype 0.0. This prototype features only basic functionality while already fitting into the concept of the framework planned for the final version. Though the capability concerning the LBM of Prototype 0.0 is limited to a standard Lid Driven Cavity scenario, the completion of this project phase is an important step. Besides the LBM-related functionality, this prototype features many helper classes and debugging possibilities that assist the programmers in further developing. Due to the detailed project management, the course of progression was almost without frictional losses but resulted in straight and fast achievement of the objectives. Furthermore, a stable foundation was laid for the whole design process of waLBerla.

9 Future Work

waLBerla will be extended according to the plans outlined in section 2. The next project part, elevating the state of the project to Prototype 0.1, will be focused on the communication between patches. While at the moment a “patch” has no representation in the code structure, the following version will support differently skilled patches. The most important part of this work is the design of the communication interface and synchronization. Since grid refinement needs more information exchange due to higher number of time steps and the need for time interpolation, the concept of communication has to be planned well and very carefully in order to enable support for later grid refinement implementation. It is planned to realize an abstract communication layer that handles communication transparently for the patch such that parallelization is already supported, too. For validation, another test case will be added, which is the “Backward Facing Step” scenario. This test case is well documented in literature such that the correctness of parametrization can be verified.

References

[CMa07] CMake, *Information on cross-platform make tool CMake*, available at http://www.cmake.org/, January 2007.

[Don04] Stefan Donath, *On optimized implementations of the lattice Boltzmann method for contemporary high performance architectures*, Bachelor’s thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2004.

[Dox07] Doxygen, *Information on source code documentation generator tool Doxygen*, available at http://www.doxygen.org/, January 2007.

[Fei06] Christian Feichtinger, *Simulation of moving charged colloids with the lattice Boltzmann method*, Master’s thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2006.

[FGID07] Christian Feichtinger, Jan Götz, Klaus Iglberger, and Stefan Donath, *Doxygen documentation on waLBerla*, January 2007.

[Göt06] Jan Götz, *Numerical simulation of blood flow with lattice Boltzmann methods*, Master’s thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2006.

[Igl03] Klaus Iglberger, *Performance analysis and optimization of the lattice Boltzmann method in 3D*, Bachelor's thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2003.

[Igl05] Klaus Iglberger, *Lattice-Boltzmann simulation of flow around moving particles*, Master's thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2005.

[KPR+05] Carolin Körner, Thomas Pohl, Ulrich Rüde, Nils Thürey, and Torsten Hofmann, *FreeWIHR: Lattice Boltzmann methods with free surfaces and their application in material science*, In: High Performance Computing in Science and Engineering, Garching 2004, Springer, 2005.

[Par07] ParaView, *Information on visualization tool ParaView*, available at http://www.paraview.org/, January 2007.

[PKW+03] Thomas Pohl, Markus Kowarschik, Jens Wilke, Klaus Ige lberger, and Ulrich Rüde, *Optimization and profiling of the cache performance of paralle l lattice Boltzmann codes*, Parallel Process. Lett. **13** (2003), no. 4, 549–560.

[Thü07] Nils Thürey, *Physically based animation of free surface flows with the lattice Boltzmann method*, Ph.D. thesis, University of Erlangen-Nuremberg, Germany, 2007.

[VTK07] VTK, *Information on VTK, the Visualization Tool Kit*, available at http://www.vtk.org/, January 2007.

[WZDH05] Gerhard Wellein, Thomas Zeiser, Stefan Donath, and Georg Hager, *On the single processor performance of simple lattice Boltzmann kernels*, Computer & Fluids **35** (2005), no. 8–9, 910–919.