Hybrid Parallelization Techniques for Lattice Boltzmann Free Surface Flows

Nils Thürey, T. Pohl and U. Rüde

University of Erlangen-Nuremberg, Institute for System-Simulation,
Cauerstr. 6, D-91054 Erlangen, Germany

In the following, we will present an algorithm to perform adaptive free surface simulations with the lattice Boltzmann method (LBM) on machines with shared and distributed memory architectures. Performance results for different test cases and architectures will be given. The algorithm for parallelization yields a high performance, and can be combined with the adaptive LBM simulations. Moreover, the effects of the adaptive simulation on the parallel performance will be evaluated.

1. Introduction

When a two phase flow involving a liquid and a gas, such as air and water, is simplified by only simulating the liquid phase with appropriate boundary conditions, this is known as a *free surface flow*. These flows a interesting for a variety of applications, from engineering and material science to special effects in movies. Our fluid simulator to solve these free surface problems uses the lattice Boltzmann method and a *Volume-of-Fluid* (VOF) model for the liquid-gas interface. Together with a turbulence model, adaptive time steps and adaptive grids it can efficiently solve complex free surface flows with a high stability. A central component is its ability to adaptively coarsen the grid for large volumes of fluid, so that these can be computed on a coarser grid and require less computations. The full algorithm is described in [7]. In the following, we will focus on its paralellization and the resulting performance.

Figure 1. Example of an adaptive free surface simulation with the VOF LBM algorithm.

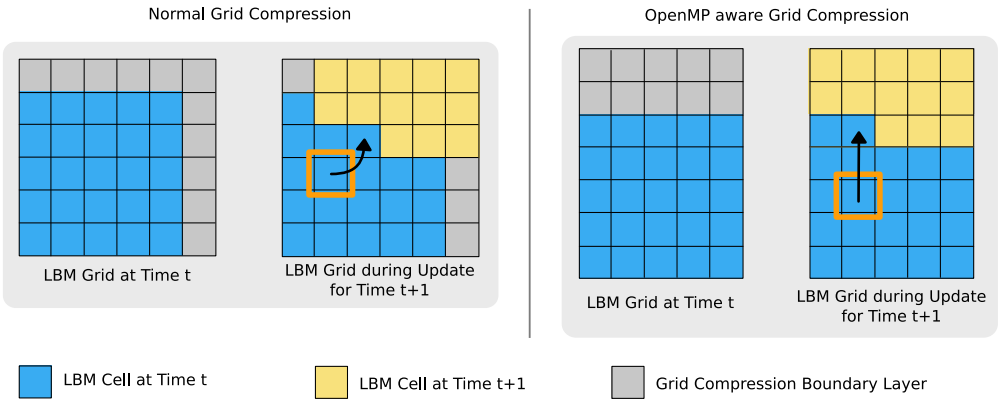

Figure 2. Comparison of the normal grid compression, and the grid compression for OpenMP. Instead of copying an updated cell to its diagonal neighbor, the cell is copied to a target cell in two cells distance (along the y direction for this 2D example, along the z direction for an actual 3D implementation).

2. OpenMP Parallelization

OpenMP is a programming model for parallel shared-memory architectures, and has become a commonly used standard for multi-platform development. An overview of the whole API is given in, e.g., [1].

The main part of the computations of the solver (ca. 73%) need to be performed for the computations of the finest grid. Thus, the parallelization aims to speed up this central loop over the finest grid of the solver. A natural way to do this would be to let the OpenMP compiler parallelize the outermost loop over the grid, usually the z direction. However, as we make use of the grid compression technique [5], this would violate the data dependencies for a cell update. With grid compression, the updated DFs of a cell at position (i, j, k) in the grid are written to the position $(i-1, j-1, k-1)$. This only works for a linear update of all cells in the grid. Instead, to use grid compression with OpenMP, the DFs of cell (i, j, k) are written back to the position $(i, j, k-2)$, as shown in Figure 2. This allows the update of all cells of an xy plane in arbitrary order. Note that this modified grid compression only requires slightly more memory than the original version (a single line of cells along the z direction, assuming the same x and y grid resolution).

Hence, the loop over the y component is parallelized, as shown in Figure 3. This is advantageous over shortening the loops along the x direction, as cells on a line along the x axis lie successively in memory. Long loops in this direction can thus fully exploit spatial coherence of the data, and prefetching techniques of the CPU if available. After each update of a plane the threads have to be synchronized before continuing with the next plane. This can be done with the OpenMP *barrier* instruction. Afterwards, the cells of the next xy plane can again be updated in any order. In the following a gravitational force along the z axis will be assumed. This usually causes the fluid to spread in the xy plane, which justifies a domain partitioning along the x and y axes.

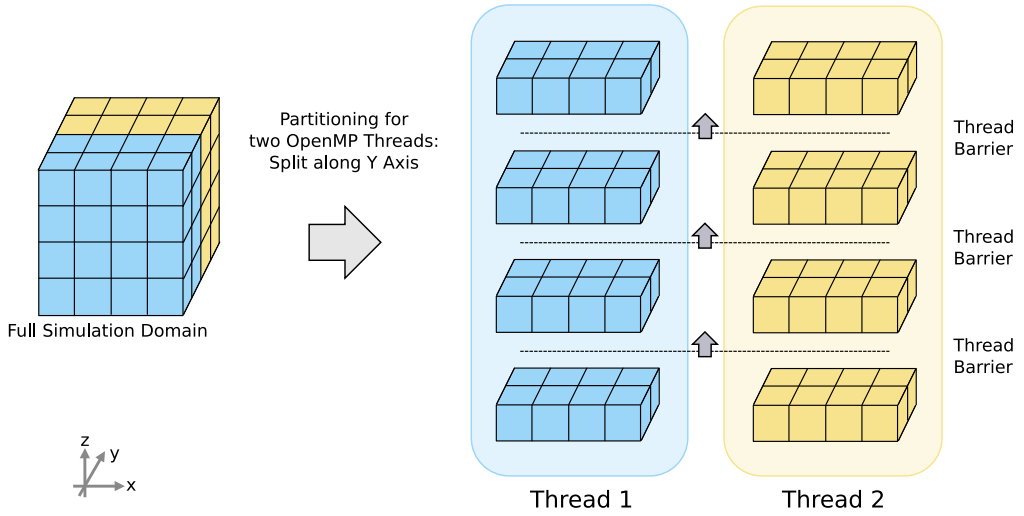

Figure 3. The OpenMP parallelization splits the y component of the loop over the grid.

OpenMP Performance Measurements

Performance measurements of the OpenMP parallelized solver can be seen in Figure 4. The graphs show absolute time measurements for a fixed number of LB steps, as *mega lattice site updates per second* (MLSUPS) measurements are not suitable for simulations with the adaptive coarsening of [7]. For these measurements, a test case of a drop falling into a basin of fluid was used. The left graph was measured on a 2.2GHz dual Opteron workstation[A], with a grid resolution of 304^3. The graph to the right of Figure 4 was measured on a 2.2GHz quad Opteron workstation[B], using a resolution of 480^3. For the dual nodes, as well as the quad nodes, the CPUs are connected by HyperTransport links with a bandwidth of 6.4 GB/s. Each graph shows timing measurements for different numbers of CPUs, and with or without the use of the adaptive coarsening algorithm.

The results without the adaptive coarsening show the full effect of the parallelization, as in this case almost 100% of the computational work is performed on the finest grid. It is apparent that the speedup is directly proportional to the number of CPUs used in this case. The four blue bars of the right graph from Figure 4 even show a speedup of 4.11 for four CPUs. This can be explained with the architecture of the quad node – the simulation setup uses most of the available memory of the machine, but each CPU has one fourth of the memory with a fast local connection, while memory accesses to the remaining memory have to performed using the HyperTransport interconnect. Thus, with four OpenMP threads the full memory bandwidth can be used, while a single thread, solving the same problem, frequently has to access memory from the other CPUs.

The timing results with adaptive coarsening show less evident speedups, as in this case only roughly 70% of the overall runtime are affected by the parallelization. As expected, the runtime for two CPUs is 65% of the runtime for a single CPU. The OpenMP parallelization thus yields the full speedup for the finest grid. It can be seen in the right graph of Figure 4 that there is a

[A] CPU: 2 x AMD Opteron 248, 2.2 GHz, 1MB L2-cache; 4GB DDR333 RAM.
[B] CPU: 4 x AMD Opteron 848, 2.2 GHz, 1MB L2-cache; 16GB DDR333 RAM.

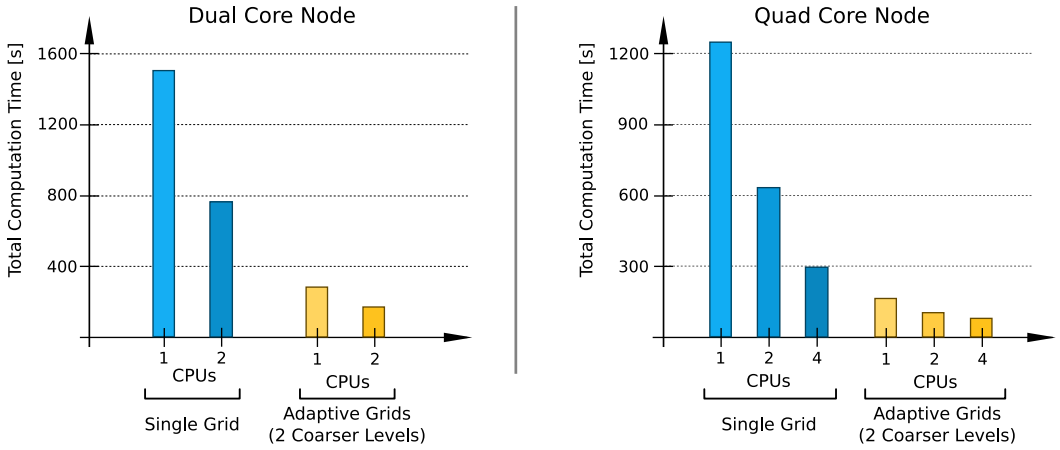

Figure 4. Time measurements for the OpenMP version of the solver: the runs to the left were measured on a dual Opteron workstation, while those to the right were measured on a workstation with four Opteron CPUs. The blue bars represent runs without the adaptive coarsening algorithm, while the orange bars use two coarse levels in addition to the finest one.

speedup factor of more than 15 between the version without adaptive coarsening running on a single CPU and the version with adaptive grids running on four CPUs. To further parallelize the adaptive coarsening algorithm, a parallelization of the grid reinitialization would be required, which is complicated due to the complex dependencies of the flag checks.

3. MPI Parallelization

For the development of applications for distributed memory machines, the *Message Passing Interface* (MPI) is the most widely used approach. In contrast to OpenMP, MPI requires more low level work from a developer, as most of its functions only deal with the actual sending and receiving of messages over the network. Details of the introductory and more advanced functions of MPI can be found, e.g., in [2] and [3].

For the MPI parallelization the domain is split along the x axis, as shown in Figure 5. In this figure two nodes are used, the domain is thus halved along the x axis, and a ghost layer is added at the interface of the two halves. Before each actual LB step, the boundary planes are exchanged via MPI, to assure valid boundary conditions for all nodes. As indicated in Figure 5, the boundary layer contains the full information from a single plane of the neighboring node. For a normal LB solver, this would be enough to perform a stream-collide-step. However, the free surface handling can require changes in the neighborhoods of filled and emptied interface cells from a previous step. All fluid cells in the layer next to the boundary layer thus have to be validated again. If one of them has an empty cell as a neighboring node, it is converted to an interface cell. This simple handling ensures a valid computation, but causes slight errors in the mass conservation, as the converted cell might have received excess mass from the former neighboring interface cell. We have found that this is unproblematic, especially for physically based animations, as the error is small enough to be negligible. For engineering applications, an additional transfer between the nodes could ensure a correct exchange of the excess mass, similar to the algorithm proposed in [6,4]. The error in mass conservation is less than 1% for

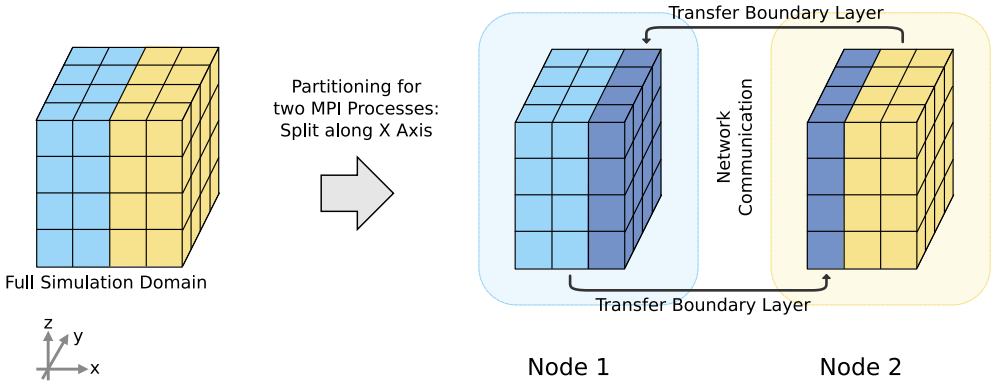

Figure 5. The MPI parallelization splits the x component of the loop over the grid.

1000 LB steps.

If this scheme is used in combination with the adaptively coarsened grids, it has to be ensured that there is no coarsening of the ghost and transfer layers. Therefore, the transfer is only required for the finest grid level. A coarsening of the ghost layers would require information from a wider neighborhood of the cells, and result in the exchange of several layers near the node boundary, in addition to the ghost layers of the different grid levels. As the bandwidth of the network is a bottleneck, such an increase of the exchanged data would result in a reduced performance. Hence, only the fine layers are connected with a ghost layer, and are treated similar to the free surface or obstacle boundaries to prevent coarsening in this region. As the node boundary has to be represented on the finest grid, this means that large volumes of fluid spanning across this boundary can not be fully coarsened. This parallelization scheme thus modifies the actual layout of the coarse and fine grids in comparison to the serial version of the solver. Further care has to be taken an adaptive resizing of the time step. This is based on the maximum velocity in the simulation, and requires the global maximum for all nodes to be computed.

4. MPI Performance Measurements and Discussion

To measure the performance of this MPI parallelized version of the solver, the test case is again that of a drop falling into a basin of fluid. The timing measurements of Figure 6 were measured on multiple quad Opteron nodes[C]. Details of these quad nodes can be found in Section 2. The x axis of each graph shows the number of nodes used for the corresponding measurement. For each node, the OpenMP parallelization of the previous section was used to execute four OpenMP threads on each node. As the parallelization changes the adaptive coarsening, Figure 6 again shows timing measurements for a fixed number of LB steps, instead of MLSUPS or MFLOPS rates. The figure shows two graphs in each row: the graph to the left was measured on a grid without adaptive coarsening, while the one to the right was measured from runs solving the same problem with two levels of adaptive coarsening. The two rows show the effect of the overhead due to MPI communication: the upper row shows results for a cubic domain of 480^3, denoted as test case Q in the following, while the lower row was measured with a wider channel and a resolution of $704 \cdot 352 \cdot 352$ (test case W). The grid resolution remains constant for any number

[C] The nodes are connected by an InifiniBand interconnect with a bandwidth of 10GBit/s

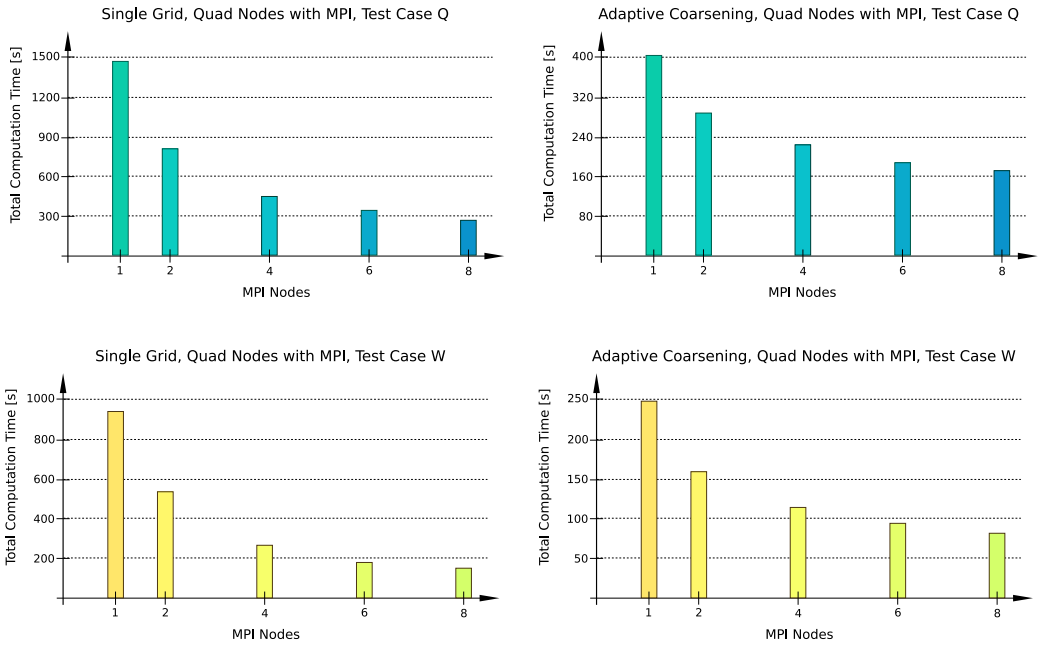

Figure 6. Time measurements for the MPI version of the solver running a problem with 480^3 (test case Q) in the upper row, and for a problem with $704 \cdot 352 \cdot 352$ (test case W) in the lower row of graphs.

of CPUs involved (strong scaling). As the domain is equally split for the number of participating MPI nodes, test case Q results in thinner slices with a larger amount of boundary layer cells to be transferred. For 8 nodes and test case Q, this means that each node has a grid resolution of $60 \cdot 480 \cdot 480$ with 480^2 boundary cells to be exchanged. Splitting the domain of test case W, on the other hand, results in slices of size $88 \cdot 352 \cdot 352$ with 352^2 boundary cells to be exchanged.

Overall, the graphs without adaptive coarsening show a speedup of around 1.8 for the strong scaling. While the speedup for test case Q, from four to eight nodes, is around 1.62, it is 1.75 for test case W, due to the better ratio between computations and communication in the latter case. This effect can also be seen for the graphs with adaptive coarsening (the right column of Figure 6). While the curve flattens out for test case Q, there is a larger speedup for test case W. For test case Q with adaptive coarsening, the speedup factor is ca. $1.3 - 1.35$, while it is between 1.45 and 1.55 for test case W. This lower speedup factor, in comparison to the test cases with only a single fine grid, is caused by the increased overhead due to MPI communication, compared to the amount of computations required for each LB step with the adaptive coarsening. Moreover, the amount of coarsening that can be performed for each slice of the grid is reduced with the increasing number of MPI processes. Practical test cases will, however, usually exhibit a behavior that is a mixture of the four test cases of Figure 6. An example of a large scale test case that was computed with four MPI processes, and required almost 40GB of memory, can be seen in Figure 8.

To evaluate the overall performance of the solver, varying rectangular grid sizes without adaptive coarsening were used to simulate problems requiring the whole memory of all participating nodes (weak scaling). While the MLSUPS rate for a single quad Opteron is 5.43 with a grid

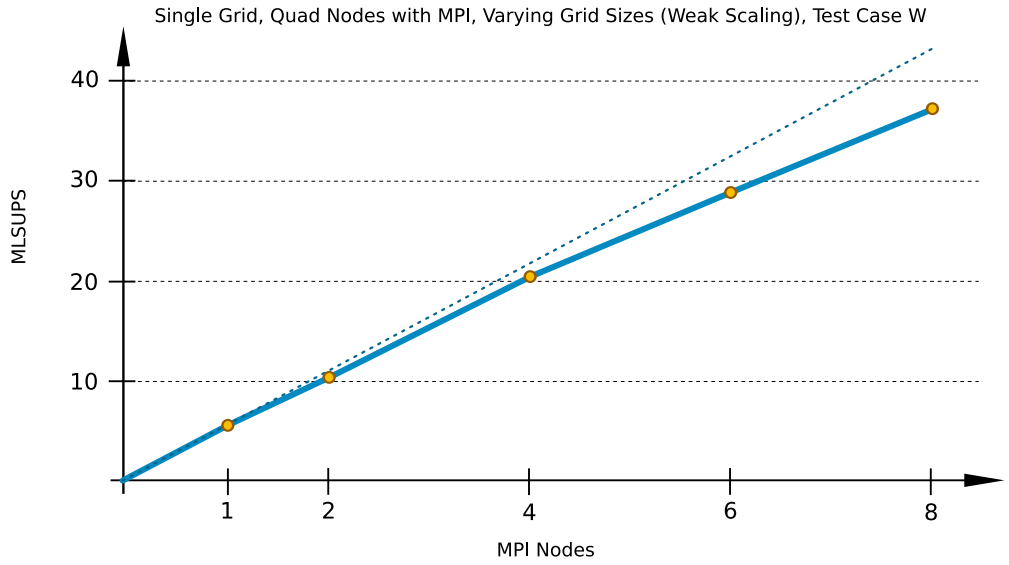

Figure 7. MLSUPS measurements for runs with varying grid resolutions (weak scaling) of test case W and without adaptive coarsening. The dotted line represents the projected ideal speedup according to the performance of a single quad node.

resolution of $704 \cdot 352 \cdot 352$, eight quad nodes with a resolution of $1040 \cdot 520 \cdot 520$ achieve a performance of 37.3 MLSUPS, as shown in Figure 7. This represents a total speedup factor of 6.87 for the eight nodes.

5. Conclusions

We have demonstrated that the parallel algorithm presented here is suitable to perform efficient large scale computations. Both algorithms for OpenMP and MPI parallelization can be combined to solve large problems on hybrid shared- and distributed-memory systems. However, the algorithm does not yield the full performance when the only goal is to reduce the computational time for small problems with MPI. For large problems, the speedup will effectively depend on the setup – for large volumes of fluid, the speedup can be around $1.3 - 1.5$, while fluids with many interfaces and fine structures can almost yield the full speedup of a factor two for each doubling of the CPUs or nodes used for the computation.

For dynamic flows, an interesting topic of future research will be the inclusion of algorithms for load balancing, e.g., those described in [4]. The algorithm currently assumes an distribution of fluid in the xy plane due to a gravity along the z direction. If this is not the case, the static and equidistant domain partitioning along the x and y axes will not yield a high performance.

REFERENCES

1. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Academic Press, 2001.
2. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, Portable Parallel Programming with the Mesage-Passing Interface*. MIT Press, second edition, 1999.

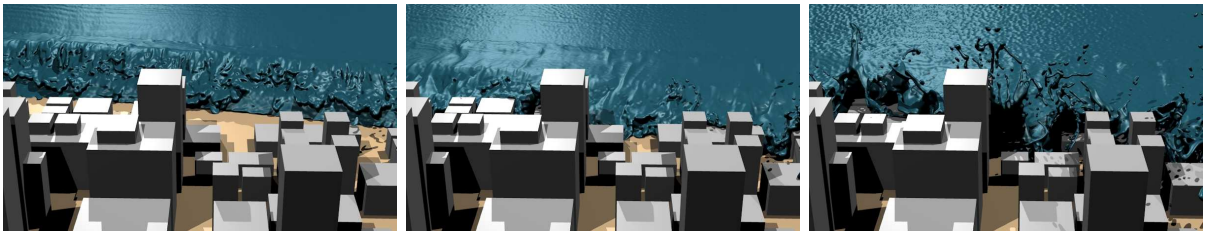

Figure 8. Pictures of a wave test case with a grid resolution of $880 \cdot 880 \cdot 336$. On average, only 6.5 million grid cells were simulated during each time step due to two levels of adaptive coarsening.

3. W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2, Advances Features of the Message-Passing Interface*. MIT Press, 1999.
4. C. Körner, T. Pohl, U. Rüde, N. Thürey, and T. Zeiser. Parallel Lattice Boltzmann Methods for CFD Applications. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *LNCSE*, pages 439–465. Springer, 2005.
5. T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes in 2D and 3D. Technical Report 03–8, Germany, 2003.
6. T. Pohl, N. Thürey, F. Deserno, U. Rüde, P. Lammers, G. Wellein, and T. Zeiser. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. In *Proc. of Supercomputing Conference 2004*, 2004.
7. N. Thürey and U. Rüde. Stable Free Surface Flows with the Lattice Boltzmann Method on adaptively coarsened Grids. *to appear in Computing and Visualization in Science, preprint available at* `www10.informatik.uni-erlangen.de/~sinithue/sfsflbmacg/cvssubmission_060529.pdf`, 2007.