Dynamic Load Balancing of Lattice Boltzmann Free-Surface Fluid Animations.

Ashley Reid
University of Cape Town
areid@cs.uct.ac.za

James Gain
University of Cape Town
jgain@cs.uct.ac.za

Michelle Kuttel
University of Cape Town
mkuttel@cs.uct.ac.za

ABSTRACT

We investigate the use of dynamic load balancing for more efficient parallel Lattice Boltzmann Method (LBM) Free Surface simulations. Our aim is to produce highly detailed fluid simulations with large grid sizes and without the use of optimisation techniques, such as adaptive grids, which may impact on simulation quality. We divide the problem into separate simulation chunks, which can then be distributed over multiple parallel processors. Due to the purely local grid interaction of the LBM, the algorithm parallelises well. However, the highly dynamic nature of typical scenes means that there is an unbalanced distribution of the fluid across the processors. Our proposed Dynamic Load Balancing strategy seeks to improve the efficiency of the simulation by measuring computation and communication times and adjusting the fluid distribution accordingly.

Categories and Subject Descriptors

I.3.7 [**Computing Methodologies**]: Computer Graphics—*Three dimensional Graphics and Realism*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms

Lattice Boltzmann Method, Parallelisation, Fluid Simulation, Load Balancing

1. INTRODUCTION

Complex fluid effects are intended to add realism to scenes in films or advertisements, resulting in a more visually-appealing and compelling experience for viewers. Due to both the large scale and the fantastical nature of many of the scenes required for competitive visual effects, there is ongoing research into effective methods for the numerical simulation of fluids for the film and animation industries.

Fluids, typically water, generate some of the most impressive and complex natural effects to be reproduced in film. However, physically recreating scenes such dramatic scenes as a city engulfed by water is not practically possible. Small-scale models are often unconvincing and expensive. Instead, media producers increasingly turn to numerical simulation. In films that are entirely computer animated, all fluids have to be realistically simulated, ranging from water, to blood, oil and even air. Air, though not visible, can effectively be treated as a fluid for simulation of its affects, such as wind, on scene objects.

For convincing simulations, it is very important to emulate the correct physical response of the fluid concerned. One approach is to fully simulate the fluid in three dimensions as accurately as possible without input from the artist. With the faster processing power of today's CPUs and improved simulation methods, such a full simulation approach is now feasible[17]. However, faster simulation speeds would allow for larger scenes to be simulated. One way to achieve this is through the use of parallel algorithms, which exploit both the multiple cores on modern CPUs and multiple CPUS in large compute farms. Such techniques have the potential to reduce simulation times from days or weeks on a single CPU to a matter of minutes or hours when running on hundreds of CPUs. Parallelisation of fluid simulations is the focus of this paper.

The contributions of this research are:

1. The first implementation of the parallel Lattice Boltzmann Method free surface simulation using dynamic load balancing.
2. A comparison of the scalability and efficiency of the static load algorithm against the dynamic algorithm, as well as a thorough analysis of the results through profiling.
3. An estimate of the number of CPUs (or processing power) required to generated fluid simulations, at the resolution used in modern movies, when using the Lattice Boltzmann Method.

2. BACKGROUND

There are a number of ways to simulate fluids and a good description of the fluid movement is needed for correct simulations. One such description is given by the Navier-Stokes (NS) equations [19, 4]. These differential equations were first solved on a grid by Harlow and Welch [15]. Foster and Metaxas then extended this to a method employing the full 3D Navier-Stokes equations [12] for computer

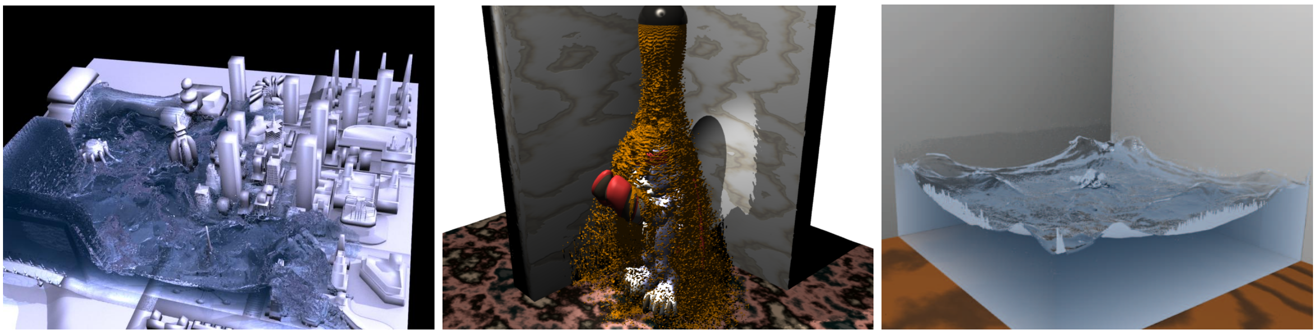


Figure 1: A single rendered frame from the City (left), Mud (centre) and drop (right) test cases.

graphics. Their method has the disadvantage of requiring very small times steps to maintain stability. Subsequently, Stam [34] introduced a superior unconditionally stable semi-Lagrangian method. It is upon this method that much of the current fluid simulation research is built [10, 11, 12, 2, 9, 17]. However, these methods still remain computationally expensive and, as a consequence, parallel implementations [17] or adaptive resolution techniques [25] have been developed in attempts to reduce the computational time required for large simulations of liquids.

Another approach to fluid simulations, Smoothed Particle Hydrodynamics (SPH), was introduced to computer graphics by Muller et al. [28]. In this method, particles are embedded into a scene and are advected using the NS equations to calculate their acceleration. Premoze et al. [30] extended the method to incorporate fluid incompressibility, while Keiser et al. [20] developed a multiresolutional approach. Losasso et al. show that the NS and SPH methods for fluid simulations could be combined [26].

The Lattice Boltzmann Method (LBM) was introduced to computer graphics by Thürey [38], based on the use of LBM in metal foaming [22, 23]. The LBM is fundamentally a cellular automaton that simulates fluid movement using a grid and sets of particle distribution functions. The distribution functions represent the movement of molecules within the fluid. Fluid is simulated by stepping these distribution functions through time using update rules. Thurey et al. [41] subsequently improved the stability of LBM simulations by introducing adaptive time steps. Later, Shankar and Sundar [33] showed that the solution obtained from the LBM simulation can be further improved with more complex boundary conditions.

As fluid simulations are computationally expensive, there has been some research into the use of parallel technologies for decreasing simulation times. In addition, there has been interest in the use of GPU accelerators for parallel fluid simulations. For example, Wei et al. [43] use the LBM on a GPU to simulate wind effects on objects such as feathers and bubbles. However, while Fan et al. [8] have shown promising real-time simulation of the underlying fluid dynamics, at the time of writing there is no known solution for real-time 3D[1] free surface flows on a GPU [37]. The LBM without a free surface has been implemented in a parallel cluster environment [6, 1, 18, 42]. Körner et al. [21] and Thürey et al. [29] provide a LBM free surface algorithm. Körner et al. [21] suggest, but do not implement, a one-dimensional slave-driven, idle process load balancing scheme for the LBM, to improve the parallel efficiency. Our purpose is to compare the efficiency of an implementation of this approach to an LBM method incorporating dynamic load balancing. A good overview of general parallel scene rendering approaches is given by Chalmers et al. [3].

3. METHOD

We use the D3D19 LBM method [16] to simulate the underlying fluid dynamics, in combination with the Volume of Fluid method [39] for extracting a free fluid surface. Our implementation uses the two simplest boundary conditions for the LBM, known as the no-slip and free-slip conditions [39, 36]. The D3D19 categorisation arises because the dynamics are simulated on a three-dimensional grid (D3), with 19 particle distribution functions (D19). The distribution functions, f_i, are shown in Figure 2. Each f_i describes the proportion of fluid for a given grid cell traveling along a specific velocity, as indicated in the figure. Fluid behaviour is achieved from the interaction of all $f_i(\overrightarrow{x}, t)$ (f_i at position $\overrightarrow{x}$, aligned with vector i) over time t.

The simulation runs in two basic steps: *Stream* and *Collide*. The *Stream* step allows $f_i(\overrightarrow{x}, t)$ to travel from position $\overrightarrow{x}$ at time t along its corresponding velocity $\overrightarrow{c_i}$ to position $\overrightarrow{x} + \overrightarrow{c_i}$ at time $t + 1$. During a given *Stream* step, each grid position is updated by copying 18 of the distribution functions to neighbouring cells. f_0 is not copied since it is the rest distribution. Two copies of the grid are needed in memory, as the distribution functions must remain consistent during the copy operation. The *Stream* step can be summarised as:

$$f_i(\overrightarrow{x} + \overrightarrow{c_i}, t + 1) = f_i(\overrightarrow{x}, t) \qquad (1)$$

Once the particles have moved to a new grid location, collisions between the new collection of distribution functions must be simulated. An example of a collision is a group of particles, f_k, moving along some none zero velocity hitting a group of particles at rest, f_0, causing some of the rest particles to gain velocity and some of the f_k to lose velocity. If the particles belong to a highly viscous fluid, such as honey, they will not change velocity as easily as a fluid with low viscosity, such as water. Collisions are resolved by relaxing each f_i towards an equilibrium distribution, which

[1] Simulations that agree with the full 3D Navier-Stokes equations.

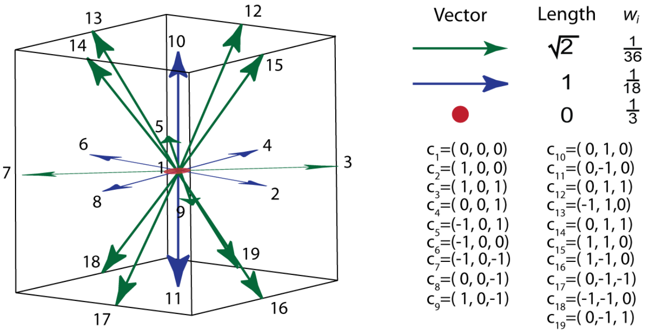


Figure 2: We employed a 3D lattice commonly known as the D3D19 lattice.The lattice has 19 different lattice velocities. Each vector represents a possible velocity along which a distribution function in the fluid can interact. These vectors allow distributions belonging to the current cell to travel to neighbouring cells. The c_1 vector gives the rest distribution function. The weights, w_i, used in the equilibrium function are provided on the right.

describes the ideal distribution of the particles along each $\overrightarrow{c_i}$ for a given fluid velocity, $\overrightarrow{u}(\overrightarrow{x})$, and density, $\rho(\overrightarrow{x})$, at a point. It is given by the following equation:

$$f_i^e = \rho w_i[1 + 3\overrightarrow{u}\overrightarrow{c_i} - \frac{3(\overrightarrow{u})^2}{2} + \frac{9(\overrightarrow{u}\overrightarrow{c_i})^2}{2}]. \qquad (2)$$

Here, the superscript e denotes the equilibrium distribution and w_i are the distribution values when $\overrightarrow{u} = 0$. Essentially, the equilibrium distribution takes the current density of the cell and distributes it along the $\overrightarrow{c_i}$ that most closely matches the fluid velocity. The density and velocity are defined as

$$\sum_i f_i = \rho, \text{ and } \sum_i f_i\overrightarrow{c_i} = \rho\overrightarrow{u}. \qquad (3)$$

At the beginning of the *Collide* step, the neighbouring distribution functions have been copied to each grid location and the density and velocity are calculated using the above equations. Then, the equilibrium values for each i are calculated and the old value for f_i is relaxed towards the equilibrium using a relaxation parameter, τ, and

$$f_i(\overrightarrow{x}, t+1) = (1 - \frac{1}{\tau})f_i(\overrightarrow{x}, t) + \frac{1}{\tau}f_i^e(\rho, \overrightarrow{u}). \qquad (4)$$

The larger the value of τ, the faster the fluid will reach the equilibrium velocity and, hence, this controls the viscosity of the simulation. In general, $\infty > \tau > \frac{1}{2}$ is required for the simulation to remain stable [31].

The Volume of Fluid (VOF) tracks the movement of the fluid by flagging grid locations as either full, empty or as interface (neither full nor empty) cells. Additionally, each cell is given a fill fraction from 0 (empty) to 1 (full). The mass loss of a cell for a given time is then calculated from the distribution functions. Interface cells can change state from full to empty. The change of the cell flags is performed in the *Process Queues* step of our implementation and presents additional complexities in the parallisation of the algorithm, as the current *Process Queues* step is dependent on the previous *Collide* step and the next stream is dependent on the current *Process Queues* step. This adds a extra point of synchronisation.

The fluid surface is then extracted by performing a reconstruction of the 0.5 fill fraction isosurface on the discrete grid. We use the Marching Cubes algorithm [24] to produce a triangulated mesh of this isosurface, which is saved to disk. The final renderings are done offline and are not part of the simulation.

3.1 Parallelization

Slice decomposition (along planes orthogonal to one of the coordinate axes) is used to divide up a scene into chunks of work for each CPU core. The Master node decides on the initial division of the scene, counting the fluid and interface cells (which account for most of the computation time) and dividing them as evenly as possible amongst the slaves. Figure 3 illustrates division of part of a scene. In this case, the division is not perfect, as it is aligned with planes of the grid and each slice may not necessarily contain the same number of fluid cells.

A given slave node in the cluster, i, will only ever communicate with the Master, slave $i-1$ and slave $i+1$, unless it is the first or the last slave in the domain. Each slave stores information on the cells for which it is responsible and two additional layers of *halo* cells (to either side), which hold the information received from a slave neighbour. The slave does no updating of the halos, but uses them to look up values during the update of its dependent cells.

The surface is constructed by each slave for the cells within its domain. This leaves a gap in the surface, so slave $n-1$ is responsible for filling the gap between slave $n-1$ and n. The Marching Cubes algorithm requires access to multiple cells in order to correctly calculate normals, so one additional plane of fill fractions beyond the halo is made available. These values are not needed for cells far from the fluid surface, so the amount of transmitted data is reduced by only sending the cells that will be used in surface construction, specifically those cells whose fill fraction, α, is opposite to an adjacent cell's fill fraction, β, in the sense that if $\alpha < 0.5$ then $\beta > 0.5$ or if $\alpha > 0.5$ then $\beta < 0.5$.

Our parallel implementation makes use of the Message Passing Interface (MPI) protocol [13] in a cluster environment. The specifications for the hardware are given in Table 1. This is a multi user system, so it was not possible to obtain guaranteed loads at any point in time. To overcome testing variations for each time recorded, 5 tests were run, but the worst two were discarded as outliers and the remaining three were averaged.

3.2 Synchronisation

The fluid simulation generates distribution functions (DFs), pressure, mass and fill fractions. In addition, the state of each cell is represented by a cell flag. Each slave is responsible for a part of the fluid domain, but requires knowledge of cell neighbours to update cells on the boundaries of its domain. Synchronisation between nodes is required and we present a design to reduce the cost of these communication calls.

Figure 4 indicates the quantities required and quantities updated in each step of our algorithm. Ideally, each of the updated data items should be synchronized after the appro-

Nodes	CPU Cores/Node	CPU Type	Interconnect	$T_{start-up}$	T_{data}	Memory/Node
160	4	Dual core 2.4Ghz Xeon	Infiniband	$87.06\mu s$	$396 Mbs^{-1}$	17GB

Table 1: The target simulation cluster. The values for $T_{start-up}$ and T_{data} were calculated using SKaMPI [32].

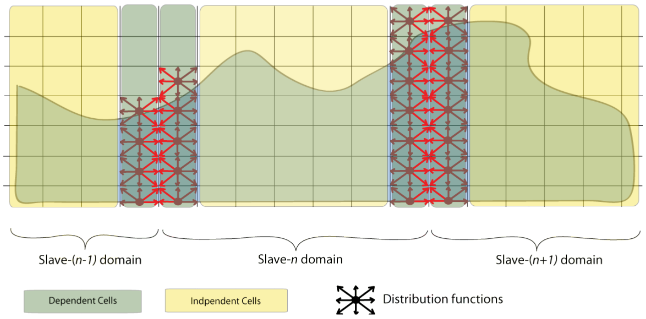


Figure 3: Dependent and independent cells. Shown here are parts of a 2D scene that have been divided up using slice decomposition. Here, the slaves $n-1, n$ and $n+1$ are shown. The dependent cells are in green on the outer edges of the domain, while the independent cells are shown in yellow on the inside of the domain. The three DFs shown in red are required by the neighbour slaves and are packed together during a later synchronisation. Other quantities required by the dependent cells are the mass, velocity and pressure.

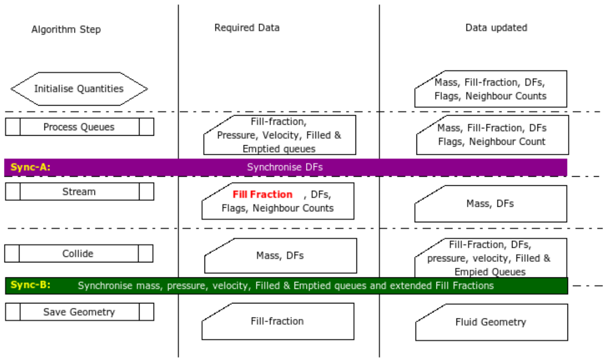

Algorithm Step
Required Data
Data updated
Initialise Quantities
Mass, Fill-fraction, DFs, Flags, Neighbour Counts
Process Queues
Fill-fraction, Pressure, Velocity, Filled & Emptied queues
Mass, Fill-Fraction, DFs Flags, Neighbour Count
Sync-A:
Synchronise DFs
Stream
Fill Fraction , DFs, Flags, Neighbour Counts
Mass, DFs
Collide
Mass, DFs
Fill-Fraction, DFs, pressure, velocity, Filled & Empied Queues
Sync-B:
Synchronise mass, pressure, velocity, Filled & Emptied queues and extended Fill Fractions
Save Geometry
Fill-fraction
Fluid Geometry

Figure 4: Simulation steps and required quantities with synchronisation points indicated (in purple and green). The required data indicates the data is used within the step, while the data updated indicates the data changed during that step. The fill fraction input into the *Stream* step has minor differences due to the Process Flag Changes step where mass is only distributed to cells within a slaves domain. Extended fill fractions are needed for normal calculations.

priate step has completed. However, this would introduce three points of synchronisation. Instead, steps are grouped according to the data they need from neighbours, as some steps, such as *Collide*, may only operate on local data. To hide the communication overhead, we overlap as much of the communication time as possible with computation arising from local data. This is implemented using non-blocking communication calls, with appropriate buffering.

The *Process Queues* operation adds additional complexities, as the *Filled* and *Emptied* queues have to be processed in the correct order to avoid erroneously emptying cells adjacent to fluid cells. This does not pose a problem for the sequential algorithm, as the entire *Filled* queue is processed before the *Emptied* queue. It is important to note that slaves only add *Filled* or *Emptied* cells to the queue for cells within their domain and the neighbour slaves must then be informed of the change in the cells on the edge of their domain. To accomplish this, additional *Filled halo* queues replicate the elements in the queue on the edge of domains.

Emptied halo queues are formed by a post-process of cells in the $x = 1$ and $x = width - 1$ planes, to identify those marked as emptied, taking care not to count cells that would be effected by the *Filled* queues. The two queues are then sent to their neighbours, as they are consistent and correctly ordered and thus usable for domain updates. The halo queues are unable, however, to inform slaves of the addition of an interface cell to the halo plane, causing inconsistencies in the cell neighbour counts. Processing of the halo queues allows each slave information on the flag state cells on the edge of the domain, in preparation for the *Stream* step.

The two points of synchronisation, Sync-A (before the *Stream* step) and Sync-B (after the *Collide* step), are shown in Figure 4. In Sync-A, the fill fractions are changed due to the redistribution of mass from filled or emptied cells in the *Process Queues* step. The difference in fill fraction is on the order of 1% of the total fill fraction, as the mass that is redistributed is only from cells that are within 1% of filling or emptying. This value is averaged with neighbouring fill fractions in the *Stream* step, so the influence on the simulation mass redistribution is negligible.

In Sync-B, the mass, pressure, velocity, filled queues, emptied queues and extended fill fractions are synchronized. The Sync-B operation also counts neighbour flags for each cell on the $x = 1$ and $x = width - 1$ planes. This is important, as the *Process Queues* step could have added new interface cells to the halos. When employing a *Stream*, *Collide* and *Process Queues* order of operations, minor inconsistencies were found in the fill fractions when saving the geometry. This is undesirable, as it leads to breaks in the fluid surface between slave geometries. To remedy this, we took advantage of the fact that the three steps are cyclic. The steps were reordered to synchronise the fluid fractions before the geometry is saved, i.e.: *Process Queues*, *Stream* and *Collide*. The geometry at this point contains no information from the filled and emptied queues, but this is only a redistribution of 1% of the fill fractions (as mentioned above). This is not an issue, as the slight behaviour difference between the fluid surface of a perfectly synchronised simulation and the optimised version is unlikely to be detectable by the human eye. After Sync-B is complete, the flags and cell neighbour counts for the cells on the edge of the domain are updated from the *Filled* and *Emptied* queues received from slave neighbours.

3.3 Static and Dynamic Load Balancing Algorithms

Thus far, we have described a system where the simulation data is passed once from the Master to a number of slaves, which then update the simulation and send the resultant fluid surface back to the Master. For each iteration, the slaves must synchronize with each other. This means that an iteration is only as fast as the slowest slave, as all the slaves must wait for that slave to complete its computation. Through load balancing, we aim to ensure that each slave runs for the same time per iteration, thereby maximising the parallel efficiency.

The simplest approach to load balancing involves a static balance of the load: dividing the scene evenly, so that each slave receives equivalent scene volumes to simulate. As fluid is accelerated due to gravity, it is possible to predict the movement of fluid downwards, so load balancing should divide the scene along an axis perpendicular to the direction of gravity. In scenes without complex floor geometry, gravity will ultimately balance the load well. A 1D decomposition helps load balancing, as it keeps the load balancing logic simple and reduces side-effects due to multiple synchronisations with neighbours. The more neighbours a slave is required to synchronize with, the more possible load connections, which makes it harder to load balance correctly. Körner et al. [21] suggested such a 1D scheme that adjusts the domain decomposition when the measured idleperiod for synchronisations with neighbouring processes is too large. At this point, a load balance is initiated and a plane of cells is fetched from the neighbour causing the wait.

In our implementation, the idleperiod while waiting for each of the Sync operations to finish, is stored separately for each slave-neighbour. If the idleperiod is larger than a threshold percentage of the total time for the current load balancing step, the slave that has been waiting requests a load balancing phase from the slave-neighbour in question, by sending a message containing the idleperiod. Upon receiving the request, the slave-neighbour will compare the received idleperiod to its current idleperiod and, if it is appropriately larger, the request will be acknowledged. In this case, the two slaves enter a load balancing phase, during which their domains are re-sized. If however, the received idleperiod is not sufficiently long, the load balancing is denied.

The load balancing phase proceeds as follows:

1. Wait for *Sync-B* and then process *Sync-B* to ensure local domain consistency.
2. Increase or decrease the current domain to cater for adjustment of domain size.
3. Send or receive the plane of VOF LBM cells resulting from the change, to or from slave-neighbour.
4. Perform the Sync-A and Sync-B operations for halo consistency.

Our implementation requires three parameters to be optimised to obtain the best results for the load balancing algorithm: the computation threshold at which a slave decides to fetch more data from a neighbour, the interval at which this is decided and the wait factor. The wait factor is how many times longer a given slave must wait compared to a neighbour's idleperiod before load balancing is allowed. This seeks to avoid cascading load balances, as large idleperiods due to an individual slave waiting for a neighbour will be ignored. Load balancing should only occur when the actual computation time causes a large idleperiod. We use a sparse set of tests to determine the optimal values for these parameters, as detailed in the results.

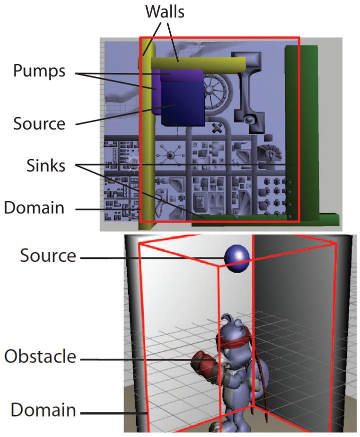


Figure 5: City (Top) and Shower (Bottom) test cases

4. TEST CASES

We compare the time taken for dynamic and static load balancing for a number of test cases at different simulation scales. The test cases used are based both upon the target applications, namely animation sequences for movies and advertisements, and common test cases employed in the literature. These test cases are (see Figure 5 for two example setups and Figure 1 for some resulting frames):

(1) Waterfall scene - A scene with a curved river bed and a pump in the upper left corner of the figure that fills the already half-filled upper section of the river with water. The water then overflows and falls to the lower level of the river. (2) Wave breaking over a city - To introduce water to the scene, a large block of water is placed in one corner of the domain, along with two pumps. This mimics the scene from *The Day After Tomorrow* where a city is flooded by a vast wave[2]. (3) Gnomon[3] showering in water/mud - This depicts a creature being showered with fluid. The inspiration for this test case is a scene in the film *Shrek*[4]. (4) Breaking dam - This the a standard falling dam example[38], with a block of water in one corner of the domain. (5) Waterdrop falling into a pool - A standard in the literature[29]: a waterdrop falls into a pool below.

Each of the test simulations were run for different numbers of CPUs. For these measurements, the wall clock time elapsed for each of the test simulations was recorded from the time the Master process started loading a scene specification until it had saved the last frame's fluid mesh. The Tuning

[2] http://www.imdb.com/title/tt0319262/
[3] A fantastical Digimon character from TV.
[4] www.shrek.com

and Analysis Utilities (TAU) [27] was used to profile each of the simulations running on different numbers of CPUs. To minimise the difference in profiled code and normal running code, only key functions were profiled, as follows. Each of the simulations were decomposed into four main sections:

1. *Stream* - This contains the LBM *Stream* and *Collide* sections and the mass transfer.

2. Communication - This section is responsible for the synchronisation of the slaves with each other and the Master (note that the idleperiods for synchronisation are included).

3. Data Packing - This encapsulates the extra work performed to prepare the data needed to synchronise the slaves.

4. Load Balance - This records the time spent load balancing.

Surface construction was not included, as it only accounts for 3.5% of the overall run-time of the simulation [31].

5. RESULTS

We present the results from the VOF LBM parallel simulations. Renderings of one frame of each of the final test cases are shown in Figure 1. These renderings were produced with Mantra[5] from SideFX, with various shaders being used for the different objects in the scene. As mentioned previously, these were generated offline and after the simulation had been saved the mesh to disk. The drop simulation used grid dimensions of $600 \times 600 \times 600$, the mud simulation of $480 \times 720 \times 480$ and the city $600 \times 165 \times 600$. In general, the fluid showed good physical fluid behaviour, when judged by the human eye, and a significant amount of detail was generated by the parallel simulations (see the amount of drops visible in the mud simulation). However, there were problems with instabilities when fluid speed became too great, as such methods of Thürey et al. [40] or d'Humiéres et al. [7] were not used. The visible effects of such instabilities can be seen in the extra number of water drops being created in the mud test case and unphysical behaviour of the drop test when speed was too great.

The parallel efficiencies of each of the simulations on varying numbers of CPU cores are shown in Figure 6, where each simulation involved 1 Master process and N slave processes. Note that, for the cluster architecture, each compute node has four physical cores and four processes can be run at a time. Thus, processes are first assigned to the same compute node so communication times are minimised. Processes on the same node make use of the memory based communication of MPI instead of using the Infiniband interconnect. The figure shows that in general the dynamic load balancing performs better for all simulations when using a small number of CPU cores. All the graphs exhibit a crossover point where the static algorithm starts performing better. The number of CPU cores at which this crossover point occurs is (approximately): Dam - 18, Drop - 25, Mud - 32, Waterfall - 14 and City - 25.

5.1 Profiling

Execution profiles for all test simulations are shown in Figure 8. The graphs show the maximum total time taken by all slaves for a given section of the simulation. It is important to consider the maximum time, as the wall clock time of the simulation will be limited by the slowest slave. Note that the computation time and communication times will be partially linked, as the longer a slave takes to update its designated grid cells, the longer another slave must wait to communicate.

6. DISCUSSION

6.1 Dynamic Load balancing parameters

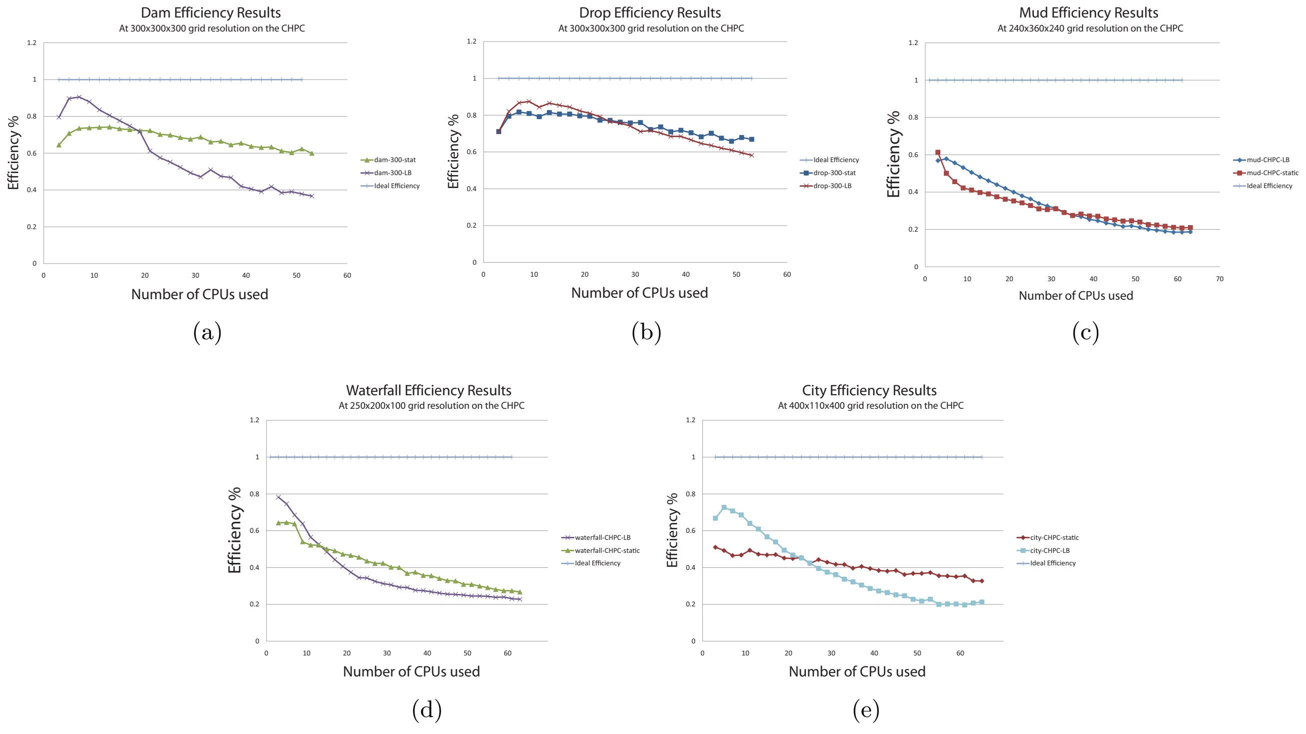
We found the computation threshold to have the greatest influence on the speed of the simulations, followed by the load balancing interval and, lastly, the wait factor. Our results show no consistent minimum for all scenes: the final choice of parameter values is a balance between making the load balancing decisions sensitive to an imbalance (which is good for simulations such as the dam) or insensitive (good for simulations such as the drop). As a design decision, we choose an load balancing interval of 1, as this allows the simulation to adjust as often as possible, favouring unbalanced simulations. The wait factor is also chosen as 1, but, as stated, does not play a large role in the load balancing decisions when compared to the effect of the computation threshold. The optimal range for the computation threshold was found to be $0.03 - 0.11$.

6.2 Profiling

Profiling (Figure 8) of the execution reveals the reasons for the differences in performance for the dynamic and static load balancing approaches. Firstly, Figure 8(a) shows that the static load balancing approach has an unbalanced load for low numbers of CPU cores and becomes far better for higher values because the maximum time for a given slave is far larger for low CPU core numbers. The data packing time remains constant, as the shared boundary between slaves does not change in size. This is one of the causes of poor scaling. For dynamic load balancing, the load for lower numbers of CPUs is balanced better. Figure 8(a) shows that ultimately the balance degrades for large numbers of CPUs. This probably occurs because slaves only have information about the time they spend waiting for their neighbours and the staggered computation times across the whole domain prevent an unbalanced slave on one end of the domain from claiming work from a slave on the other end. For example, consider a simulation with 10 slaves. Let slave 2 have computation time of x seconds per iteration. Now, let slave 3 have a computation time of $0.95x$ and slave 4, $(0.95)^2x$, etc. Eventually slave 8 will have a computation time of $0.66x$, which is a definite candidate for load balancing, but the difference with its neighbours is only 5% in terms of computation time, so it will not ask to load balance.

In the end, Figure 8 shows that the communication time, although initially small relative to the *Stream* time, eventually becomes much more comparable. This causes the poor scaling, as it forms a large portion of the effectively non-parallelizable time (which is constant for all slaves). The other factor causing poor scalability of the dynamic simulation is the time required to load balance, which increases

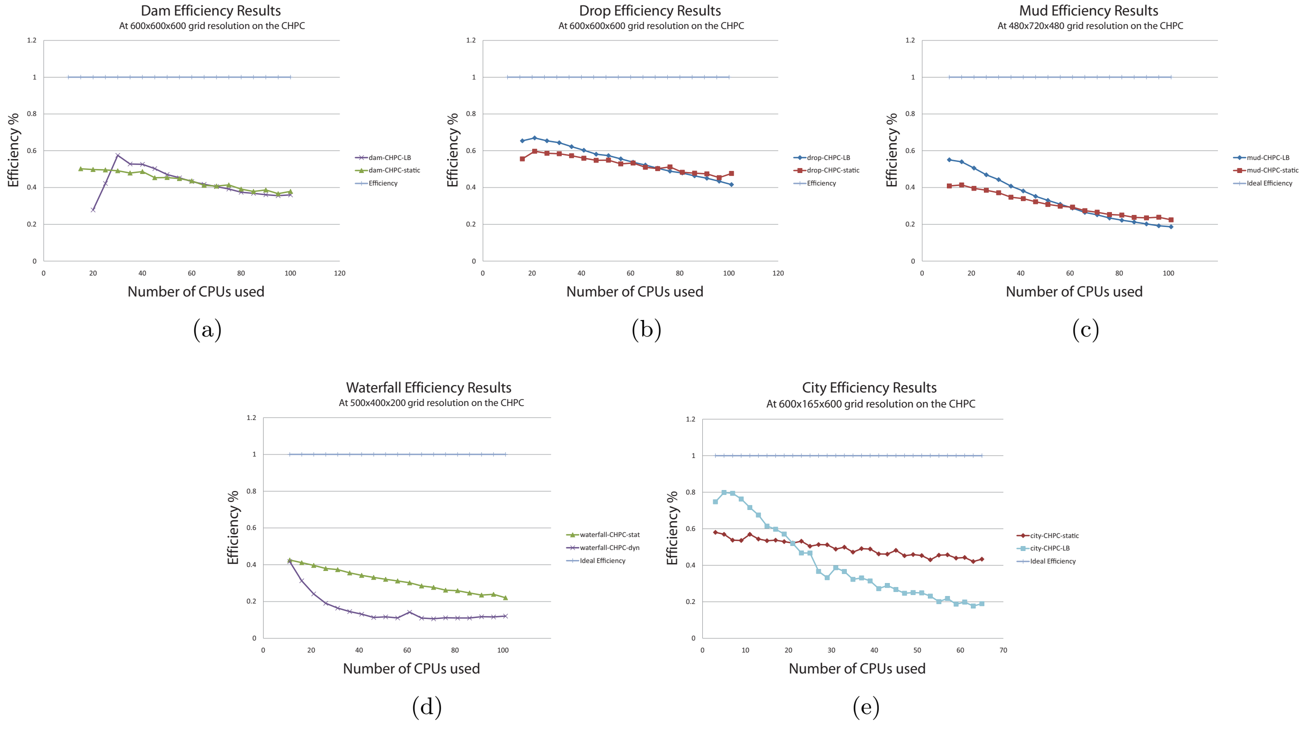
[5] www.sidefx.com



(a) (b) (c)

(d) (e)

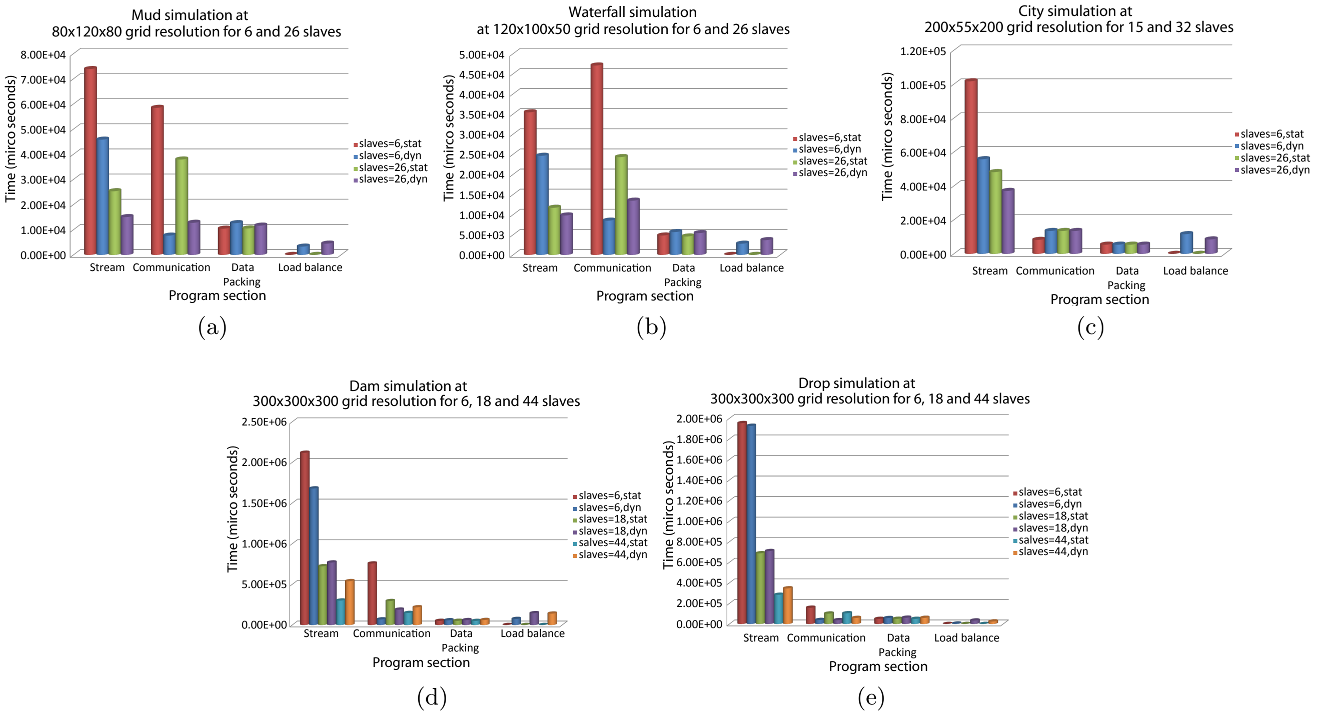
Figure 6: The efficiency results for the test cases at a medium resolution. The label CPU refers to a CPU core.



(a) (b) (c)

(d) (e)

Figure 7: The efficiency results for test cases at high resolutions.The strange "kink" in the dynamic load balancing graph, for the dam simulation, is due to the initial conditions of the scene being set the same as the static load balancing. The dynamic initial conditions require larger memory, causing virtual memory thrash. The label CPU refers to a CPU core.



(a)

(b)

(c)

(d)

(e)

Figure 8: Profiling of simulations. The four major sections of the simulation have been profiled and are indicated on the graphs. (a) Mud, (b) Waterfall, (c) City, (d) Dam and (e) Drop test cases. Each slave runs on its on CPU core.

with the number of CPUs. With higher numbers of CPUs, it becomes harder to determine exactly when a load balance should occur, as the time for each iteration is reduced and the relative idleperiods are higher. Thus, the system becomes more sensitive to load imbalances, causing an increased number of load balances and the increase in time on the graph.

Figure 8(e) shows the profiling graph for the drop simulation. It exhibits very similar characteristics, except the balance in general is far better. Similar effects are observed for lower resolutions, but the main difference is that the ratio of computation to communication is lower, causing even poorer scaling.

The mud and water simulation exhibit very poor scaling, for all resolutions. From 8(a) and (b) it is easy to see that the reason for such behaviour is that the communication time outweighs the computation time. In addition, the packing time and the load balancing time reduces the scaling. The scene specification has a small amount of fluid and this is the cause of the small streaming time. If streaming is proportionally small, then there will be poor scaling as there is less total computation to be distributed among the slaves. The ratio of computation to communication is low. The higher resolution scaling exhibits similar effects.

The city simulation in 8(c) has a static case that is very unbalanced, in comparison to the dynamic case. This is understandable when the scene specification is considered. The scene initially has large empty regions that have no fluid, which is an easy mistake to make, as an artist would not always be aware of the optimal scene specification. This will adversely affect the static load balancing algorithm, as it has no way of catering for such a scenario. Hence, dynamic load balancing performs better for this test case, as it measures such an imbalance and adjusts slave-domains accordingly.

6.3 Scalability and Efficiency

For simulations at different resolutions, the scalability of the VOF LBM simulation is good for low numbers of CPUs, but eventually the communication prevents effective scaling, as can be seen from the profiling graphs. For the lower numbers of CPUs, where the scaling is good, dynamic load balancing improves the efficiency of the algorithm. In most cases, the dynamic load balancing performs worse for larger numbers of processors, as a small mistake in the load balancing can cause one node to have a higher total time, thus slowing the entire simulation. As mentioned above, this is because only local slave computation times are considered during balancing. Incorporating a higher order scheme that regulates the load balance with global knowledge could be beneficial. In addition, the dynamic load balancing is also affected by the extra communication needed to perform load balancing. This unbalances the communication-to-computation ratio even further.

Overlapping the communication and computation helps to explain the better scaling for lower numbers of CPUs as most of the communication time can be hidden. However, the time required for communication and data packing remains constant for all numbers of CPUs and so eventually dominates the simulation run time.

A positive aspect of the scalability results at different resolutions is that the system exhibits good "weak scaling". Weak scaling measures the system performance against number of CPUs as the problem size (in our case, the size of the grid) increases, proportionally to the number of CPUs [5]. Conversely, strong scaling fixes the problem size. A higher number of total grid cells, means a higher number of cells assigned to each slave and the ratio of computation to communication becomes larger. Still, for the mud and waterfall simulations, with smaller amounts of fluid, the scaling is not as good as the drop and dam simulations. This result indicates that parallel VOF LBM is suited to large scenes with a high proportion of fluid, but will still have poor scaling for scenes with low amounts of fluid agreeing with Gustafason's law [14].

It is important to note that the VOF LBM algorithm requires a large amount of communication every iteration. Therefore, good scaling is not expected. With this in mind, the scaling results for our implementation are positive.

6.4 Architecture and scene scale recommendations

The size of animations currently created in research and, hence, movies is similar to that of our high resolution tests (see resolutions in Losasso et al. [26]). Here we use the scaling at these resolutions to make recommendations on the number of CPUs for creating simulations. A required reference duration for a scene is chosen to be 1 min (180 frames). In reality, the scene could be very short, or possibly longer, but this is a suitable length of time to draw conclusions. The film duration for drop and dam test cases is 3.5s and the duration of the mud, waterfall and city is 7s. Essentially, the reference time is 17 times longer than the simulations produced.

The time it would take to produce 1 min of similar simulations, ranges from 3hrs for the waterfall simulation to 17hrs for the drop and dam simulation, when using 35 CPUs. When using 100 CPUs, these times become 2hrs and 8.5hrs. At these resolutions, there is still room for scaling at reasonably efficient rates, as the simulation is not yet dominated by the communication. Fitting a power regression line to the static load balancing times of the drop and dam yields an estimate of 4.5hrs when using 200 CPUs. This time is more acceptable, as this would allow the creation of 2 simulations within a 9hr day. In production this would mean that a simulation with one edit could be made within a day. At these levels the returns are diminishing with increasing numbers of CPUs. For this number of CPUs, further optimisations of communication are required, as that forms the main bottleneck.

7. CONCLUSIONS

We have presented a parallel algorithm for Lattice Boltzmann Free-Surface Fluid Dynamics suitable for implementation on a cluster of CPUs. In particular, we focus on dynamic load balancing in order to ensure an equal distribution of computation between nodes during the simulation.

In general, parallelisation of fluid simulation is highly desirable since with high numbers of CPUs it is possible to reduce typical simulation times from around 17 hrs to 8.5 hrs or less. Further, our tests show that dynamic load balancing outperforms static load balancing only for small numbers of CPUs, typically less than 20-30 CPUs. The cross-over point depends on the nature of the simulation, with dynamic load balancing favouring large scenes with a high proportion of fluid.

There are several viable extensions to this work, most notably parallelisation across a mixed cluster of CPUs and GPUs. More complex global dynamic load balancing schemes, that take into account the load across the entire system, may show benefits. It would also be worth including dynamic objects that interact with the fluid simulation, as this is often a requirement for film.

8. REFERENCES

[1] G. Amati, S. Succi, and R. Piva. Massively Parallel Lattice-Boltzmann Simulation of Turbulent Channel Flow. *International Journal of Modern Physics C*, 8:869–877, 1997.

[2] M. Carlson, P. J. Mucha, and G. Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3):377–384, 2004.

[3] A. Chalmers, T. Davis, and E. Reinhard. *Practical parallel rendering*. AK Peters, Ltd., 2002.

[4] J. X. Chen and N. da Vitoria Lobo. Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equations. *Graph. Models Image Process.*, 57(2):107–116, 1995.

[5] H. Dachsel, M. Hofmann, and G. Raunger. Library support for parallel sorting in scientific computations. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 695–704. Springer Berlin Heidelberg, 2007.

[6] J.-C. Desplat, I. Pagonabarraga, and P. Bladon. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. *Computer Physics Communications*, 134:273–290, Mar. 2001.

[7] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Phil. Trans. R. Soc. A*, 360:437–451, 2002.

[8] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society Washington, DC, USA, 2004.

[9] R. Fattal and D. Lischinski. Target-driven smoke animation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 441–448, New York, NY, USA, 2004. ACM.

[10] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM.

[11] N. Foster and R. Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, New York, NY, USA, 2001. ACM Press.

[12] N. Foster and D. Metaxas. Realistic animation of liquids. *Graph. Models Image Process.*, 58(5):471–483, 1996.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.

[14] J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[15] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[16] F. J. Higuera, S. Succi, and R. Benzi. Lattice gas dynamics with enhanced collisions. *Europhysics Letters*, 9:345–+, June 1989.

[17] G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 805–811, New York, NY, USA, 2006. ACM Press.

[18] D. Kandhai, A. Koponen, A. G. Hoekstra, M. Kataja, J. Timonen, and P. M. A. Sloot. Lattice-Boltzmann hydrodynamics on parallel systems. *Computer Physics Communications*, 111:14–26, June 1998.

[19] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 49–57, New York, NY, USA, 1990. ACM Press.

[20] R. Keiser, B. Adams, L. J. Guibas, P. Dutri, and M. Pauly. Multiresolution particle-based fluids. Technical report, ETH CS, 2006.

[21] C. Körner, T. Pohl, , N. Thürey, and T. Zeiser. *Parallel Lattice Boltzmann Methods for CFD Applications*, volume 51. Springer Berlin Heidelberg, 2006.

[22] C. Körner and R. F. Singer. Processing of metal foams| -| challenges and opportunities. *Advanced Engineering Materials*, 2(4):159–165, 2000.

[23] C. Körner, M. Thies, and R. Singer. Modeling of metal foaming with lattice boltzmann automata. *Advanced engineering materials(Print)*, 4(10), 2002.

[24] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[25] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 457–462, New York, NY, USA, 2004. ACM.

[26] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw. Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[27] B. Mohr, D. Brown, and A. Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 29–29, 1994.

[28] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[29] N. T. T. Pohl and U. Rüde. Hybrid Parallelization Techniques for Lattice Boltzmann Free Surface Flows. *International Conference on Parallel Computational Fluid Dynamics*, May 2007.

[30] S. Premoze, T. Tasdizen, J. Bigler, A. Lefohn, and R. Whitaker. Particle-based simulation of fluids. *Eurographics 2003*, 22(3), 2003.

[31] A. Reid. Parallel fluid dynamics for the film and animation industries. Master's thesis, University of Cape Town, 2009.

[32] R. Reussner. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[33] M. Shankar and S. Sundar. Asymptotic analysis of extrapolation boundary conditions for lbm. *Comput. Math. Appl.*, 57(8):1313–1323, 2009.

[34] J. Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[35] M. Sturmer, J. Gotz, G. Richter, and U. Rude. Blood flow simulation on the cell broadband engine using the lattice boltzmann method. Technical report, Technical Report 07-9. Technical report, Department of Computer Science 10 System Simulation, 2007.

[36] S. Succi. *The Lattice Boltzmann Euqation for fluid dynamics and beyond*. Oxford University Press, New York, 2001.

[37] N. Thuerey. Free surface flows with lbm, October 2008. http://www.vgtc.org/PDF/slides/2008/visweek/tutorial8_thuerey.pdf.

[38] N. Thürey. A single-phase free-surface lattice-boltzmann method. Master's thesis, FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG, 2003.

[39] N. Thurey. *Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method*. PhD thesis, Technischen Fakultat der Universitat Erlangen-Nurnberg, 2007.

[40] N. Thurey, C. Korner, and U. Rude. Interactive Free Surface Fluids with the Lattice Boltzmann Method. Technical report, Technical Report 05-4. Technical report, Department of Computer Science 10 System Simulation, 2005, 2005.

[41] N. Thürey, T. Pohl, U. Rüde, M. Öchsner, and C. Körner. Optimization and stabilization of LBM free surface flow simulations using adaptive parameterization. *Computers and Fluids*, 35(8-9):934–939, 2006.

[42] J. Wang, X. Zhang, A. G. Bengough, and J. W. Crawford. Domain-decomposition method for parallel lattice boltzmann simulation of incompressible flow in porous media. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 72(1):016706, 2005.

[43] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman. Blowing in the wind. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 75–85, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.