The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Parallel fluid dynamics for the film and animation industries

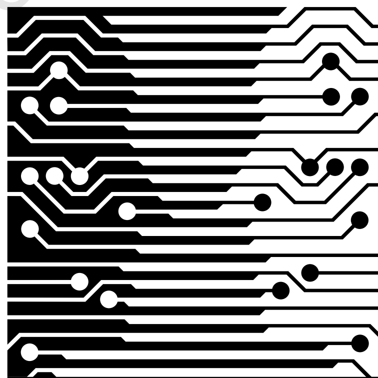
A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By Ashley Reid[1]

Supervised by: James Gain[2] and Michelle Kuttel[3]

Submitted on 18th May 2009

University of Cape Town



[1] ashreid@gmail.com
[2] jgain@cs.uct.ac.za
[3] mkuttel@cs.uct.ac.za

Abstract

The creation of automated fluid effects for film and media using computer simulations is popular, as artist time is reduced and greater realism can be achieved through the use of numerical simulation of physical equations. The fluid effects in today's films and animations have large scenes with high detail requirements. With these requirements, the time taken by such automated approaches is large. To solve this, cluster environments making use of hundreds or more CPUs have been used. This overcomes the processing power and memory limitations of a single computer and allows very large scenes to be created. One of the newer methods for fluid simulation is the Lattice Boltzmann Method (LBM). This is a cellular automata type of algorithm, which parallelizes easily. An important part of the process of parallelization is load balancing; the distribution of computation amongst the available computing resources in the cluster. To date, the parallelization of the Lattice Boltzmann method only makes use of static load balancing. Instead, it is possible to make use of dynamic load balancing, which adjusts the computation distribution as the simulation progresses.

Here, we investigate the use of the LBM in conjunction with a Volume of Fluid (VOF) surface representation in a parallel environment with the aim of producing large scale scenes for the film and animation industries. The VOF method tracks mass exchange between cells of the LBM. In particular, we implement the new dynamic load balancing algorithm to improve the efficiency of the fluid simulation using this method. Fluid scenes from films and animations have two important requirements: the amount of detail and the spatial resolution of the fluid. These aspects of the VOF LBM are explored by considering the time for scene creation using a single and multi-CPU implementation of the method. The scalability of the method is studied by plotting the run time, speedup and efficiency of scene creation against the number of CPUs. From such plots, an estimate is obtained of the feasibility of creating scenes of a giving level of detail. Such estimates enable the recommendation of architectures for creation of specific scenes.

Using a parallel implementation of the VOF LBM method we successfully create large scenes with great detail. In general, considering the significant amounts of communication required for the parallel method, it is shown to scale well, favouring scenes with greater detail. The scalability studies show that the new dynamic load balancing algorithm improves the efficiency of the parallel implementation, but only when using lower number of CPUs. In fact, for larger number of CPUs, the dynamic algorithm reduces the efficiency. We hypothesise the latter effect can be removed by making using of centralized load balancing decision instead of the current decentralized approach. The use of a cluster comprising of 200 CPUs is recommended for the production of large scenes of a grid size 600^3 in a reasonable time frame.

University of Cape Town

Acknowledgments

In the end, this was a difficult Masters. I discovered that writing is hard, more precisely, being exact with these things called “words” is hard. I would like to thank my supervisors, James Gain and Michelle Kuttel, for helping with this particular endeavor and of course the rest of the effort that they put into the work resulting in this document. Thank you to Duncan Clough for his proof reading.

Life and this thesis would be pointless if it were not for the people around me. I would like to thank my parents, Graham and Shez, for their tireless efforts in providing for me and putting up with my comings and goings. The group that was particularly supportive in the mad days before I left for Germany: Fritz (for late night DOTA), Kath (for boundless energy), Mary (for late night work parties), Ingrid (for always smiling), Duncs (for all the politics and exercise), Neil (for the chats over coffee and Chinese), Melch (for the guitar) and Lindsay (for the positivity). To various friends who supported me during this task: Sally-Anne (for energy and ’life’ support), Kers (for big sister type advice) and Jan (for squash and perspective).

The people on campus: Thyla (for much tea/coffee and philsophical discussion, in truth I owe greater gratitude for the Maths degree), Chris de Kadt for always speaking his mind, causing trouble and not accepting second rate thought), Carl (for the NVIDIA leg up and all the work into UCT computer science), Andrew (for many laughs and always improving our lives as students), Dave (for joining forces for our publication and of course for gyming), Ian (for being taller than me and for the braai’s), Jannie (for the expert DOTA technique) and Bertus (for still being there).

I would to thank the Centre for High Performance Computing in Cape Town for the use of their machines, they filled a great need and without that the thesis would have been delayed much longer. They have a wonderful attitude towards developing research in South Africa. In particular, I would like to thank Sebastian Wyngaard for his help in setting up this connection.

I would like to thank NVIDIA for the donation of the graphics cards to the lab, which aided in the development process, and for their involvement at UCT. They have given me and many the opportunity of overseas exposure and extra funding through internships.

With hesitancy, I thank God for creating a world that is very complex. I still haven’t decided if this is a good thing or not. Hopefully, this thesis explains a part of it correctly.

University of Cape Town

Notation

Here, we give brief definitions of the symbols used in the derivations and proofs. The definitions are given in three dimensions, but are similarly defined in two dimensions.

Scalars, Vectors and functions

Vectors are given using the arrow notation and scalars without. For example, the scalar pressure is be written as P and the velocity is $\overrightarrow{u} = (u_x, u_y u_z)$, with co-ordinates of velocity given by u_x,u_y and u_z. The vector $\overrightarrow{x} = (x, y, z)$ is used to represent the position vector in three dimensions.

Functions may map to a scalar value or a vector value. In the previous example, P may be a function of position, x, y and z. This is denoted as $P(x, y, z)$, or $P(\overrightarrow{x})$ in short form. An example of a vector valued function is the velocity at any point in space, $\overrightarrow{u}(\overrightarrow{x}) = (f(\overrightarrow{x}), g(\overrightarrow{x}), h(\overrightarrow{x}))$, where f, g and h are functions mapping from $\mathbb{R}^3 \to \mathbb{R}$. Often, the inputs to a function will not be included, i.e.$\overrightarrow{u}(\overrightarrow{x})$ is written simply as $\overrightarrow{u}$.

Partial Derivatives

$\frac{\partial f}{\partial x}$ denotes the partial derivative of f with respect to x, where f is a scalar valued function. This is obtained by assuming f to be constant with respect to all other variables besides x.

The gradient of a scalar function is given by

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) = \frac{\partial f}{\partial \overrightarrow{x}}.$$

Again, for scalar f, the divergence operator is

$$\nabla \cdot f = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} + \frac{\partial f}{\partial z}.$$

Now, $\frac{\partial \overrightarrow{u}}{\partial x}$ for a vector valued function is calculated co-ordinate wise, $\frac{\partial \overrightarrow{u}}{\partial x} = \left(\frac{\partial f}{\partial x}, \frac{\partial h}{\partial x}, \frac{\partial g}{\partial x} \right)$. The gradient of a vector function is called the Jacobian and is defined as

University of Cape Town

$$\nabla \overrightarrow{u} = \nabla(f, g, h) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial h}{\partial x} & \frac{\partial g}{\partial x} \\ \frac{\partial f}{\partial y} & \frac{\partial h}{\partial y} & \frac{\partial g}{\partial y} \\ \frac{\partial f}{\partial z} & \frac{\partial h}{\partial z} & \frac{\partial g}{\partial z} \end{pmatrix}.$$

The Laplacian is given as

$$\nabla^2 \overrightarrow{u} = \frac{\partial^2 \overrightarrow{u}}{\partial^2 x} + \frac{\partial^2 \overrightarrow{u}}{\partial y^2} + \frac{\partial^2 \overrightarrow{u}}{\partial z^2}.$$

And, finally, the curl of the velocity field, $\overrightarrow{u}$, is

$$\nabla \times \overrightarrow{u} = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right).$$

Tensors

The tensor notation, $u_\alpha u_\beta$, is used on occasion to indicate the sum of all combinations of co-ordinates of $\overrightarrow{u}$, i.e. $\alpha, \beta = 1, 2, 3$.

The delta function, $\delta_{\alpha\beta}$, is the indicator function, equal to 1 if $\alpha = \beta$ and 0 otherwise

University of Cape Town

Contents

1 Introduction **12**

1.1 Motivation . . . 13

1.2 Aims and Evaluation . . . 14

1.3 Contributions and results . . . 16

1.4 Thesis Structure . . . 17

2 Background **18**

2.1 Computer Simulation . . . 18

2.2 Fluid Simulation . . . 20

2.3 First Approaches and the Navier-Stokes Equations . . . 21

2.4 Simulation Methods . . . 23

2.5 The Fluid Surface . . . 31

2.6 Parallel Programming . . . 33

2.7 Using High Performance Computing . . . 37

2.8 Conclusion . . . 38

3 LBM Fluid Simulation **40**

3.1 Full Algorithm Description for 3D . . . 40

3.2 Physical Derivation . . . 44

3.3 Conclusion . . . 51

4 Single CPU Implementation **52**

4.1 Algorithm Summary . . . 52

4.2 Volume of Fluid Method . . . 53

University of Cape Town

4.3 Fluid Surface Generation 56
4.4 Houdini Integration 58
4.5 Animation Results 63
4.6 Conclusion 74

5 Parallelization **75**
5.1 Introduction 75
5.2 Design 76
5.3 Implementation 80
5.4 Conclusion 88

6 Evaluation **90**
6.1 Method 90
6.2 Results 94
6.3 Conclusion 113

7 Discussion **116**
7.1 Implementation and development 116
7.2 Animation Creation 117
7.3 Dynamic Load balancing parameters 119
7.4 Scalability and Efficiency 119
7.5 Architecture and scene scale recommendations 121

8 Conclusions **123**
8.1 Thesis Aims 123
8.2 Recommendations and Future work 124

A Advanced Fluid Effects **126**

B Proof of physical correctness **128**

C Single CPU Profiles **133**
C.1 TAU profile 133
C.2 Operation Counts 134

University of Cape Town

D Extended Tests and Results **135**

D.1 Latency and bandwidth values for CRUNCH and CHPC 135

D.2 Simulations times 139

List of Figures

2.1 Two triangulations of a 2D sphere. (a) low resolution (b) high resolution. 19

2.2 The Marker-And-Cell (MAC) grid and the markers used in the 2D MAC grid method. 23

2.3 The HPP lattice 27

2.4 Fluid simulated using Smoothed Particle Hydrodynamics (SPH). 30

2.5 Irregular domain decomposition. 36

3.1 The 3D lattice that we use is commonly known as the D3D19 lattice. 41

3.2 Distribution functions and obstacles. 42

3.3 The 2D lattice. 48

4.1 (left) The Volume of Fluid (VOF) LBM algorithm steps. (right) Scene cell division. 53

4.2 The marching cubes algorithm illustrated. 57

4.3 The Interpolated Vertices (IVs) structure used to optimise the marching cubes algorithm. 58

4.4 Pseudocode for the marching cubes algorithm used to generate the fluid surface from the VOF fill fractions. 59

4.5 The basic Houdini interface. 61

4.6 An example of the dynamic fluid network. 61

4.7 The parameters pane for the VOF LBM custom operator. 62

4.8 The falling drop at different spatial resolutions ($\Delta t = 0.05s$, $S = 0.03$ and $G = 50 \times 50 \times 50$). . . 65

4.9 The falling drop for different Smagorinsky constants ($\Delta t = 0.01s$, $\Delta x = 0.0004$ and $G = 50 \times 50 \times 50$). 66

4.10 The falling drop for different time steps ($\Delta x = 0.0004m$, $S = 0.03$ and $G = 50 \times 50 \times 50$). 67

4.11 The falling drop for different grid resolutions ($v = 0.00001ms^{-1}$ and $S = 0.03$). 68

4.12 The falling drop for different viscosities ($\Delta t = 0.05s$, $\Delta x = 0.0004m$, $S = 0.03$ and $G = 50 \times 50 \times 50$). 69

University of Cape Town

4.13 Artifacts due to the grid representation at a resolution of 80x100x80. 70
4.14 The time taken to produce the 3.6s falling drop animation for different resolutions. 71
4.15 Percentage of time spent in each of the steps during a simulation. 71
4.16 Still pond test case. 72
4.17 A analysis of the operations per cell in the stream and collide steps for the pond test case. 72
4.18 Memory usage of the VOF LBM implementation. 73

5.1 Domain decomposition methods. 77
5.2 The 2D domain decomposition viewed from the top. 79
5.3 Dependent and independent cells. 80
5.4 Simulation steps and required quantities. 82
5.5 Simulation steps and required quantities with synchronisation. 83
5.6 Flow diagram for each slave process. 85

6.1 The five tests cases used to evaluate the parallel VOF LBM implementation. 91
6.2 The final rendered version of the water fall scene. 95
6.3 The final rendered version of the water drop falling into a pool. 96
6.4 The final rendered version of the shower scene, with fluid rendered as mud. 97
6.5 The final rendered version of the wave breaking over the city. 98
6.6 The final rendered version of the breaking dam. 99
6.7 Dam simulation for different parameter values on CRUNCH. 101
6.8 Drop simulation for different parameter values. 102
6.9 The run-time for the drop and dam simulations for different computation thresholds. 102
6.10 Speedup results for all test cases. 104
6.11 Efficiency results for low simulation resolutions. 105
6.12 The speedup results for the test cases at a medium resolution.. 107
6.13 The efficiency results for the test cases at a medium resolution. 108
6.14 The speedup results for the tests cases at high resolutions. 110
6.15 The efficiency results for test cases at high resolutions. 111
6.16 Maximum and Mean Geometric deviation for the Gnomon in mud simulation. 112

University of Cape Town

6.17 Maximum Geometric deviation for static load balancing, represented by a rainbow colour scale from blue to red projected onto the single CPU model. 112

6.18 Profiling of simulations at low resolutions. 114

6.19 Profiling of drop and dam simulations at medium resolutions. 115

C.1 A profile of the single CPU implementation. 133

D.1 The `Pingpong_send_recv` times using SKaMPI. 136

D.2 Wallclock time for simulations on the CHPC and CRUNCH architectures. 137

D.3 Speedup and efficiency results for dam for a grid resolution of $100 \times 100 \times 100$. 138

D.4 Speedup and efficiency results for dam for a grid resolution of $100 \times 100 \times 100$. 138

D.5 Wall clock time for simulations at medium resolutions 139

D.6 Wall clock time for simulations at medium resolutions 140

D.7 Wall clock time for simulations at high resolutions 141

University of Cape Town

List of Tables

4.1 Modified mass exchanges to reduce lone interface cells. 56

4.2 The space taken up by one cell in the VOF LBM implementation. 71

5.1 The two different clusters on which we ran our simulations. 87

6.1 The tests that will be run to determine the scalability of the system. 93

A.1 An overview of the advanced fluid methods that have been researched by different authors. . . . 127

C.1 The operations used during the stream and collide steps of the LBM for the still pond test case. . 134

University of Cape Town

Chapter 1

Introduction

Directors seek to produce films or advertisements with complex fluid effects that add realism to scenes and, ultimately, a more visually pleasing experience to viewers. Due to the large scale and fantastical nature of many of the scenes required for competitive visual effects, there has been a significant body of research into creating fluid using computer simulations. These methods have been helped by the increase in CPU power and parallel systems that use architectures such as clusters. The end result is effective fluid simulation for the film and animation industries.

Computer simulations use many methods to approximate fluid, often making use of physical equations to describe fluid behaviour. Algorithms are then created using these equations to produce fluid effects. The fluid is used as it is or is used together with other visual elements, such as adding water to a city filmed from above. One of the newer and less tested algorithms for fluid simulation is the Lattice Boltzmann method. This thesis adds to the research on the Lattice Boltzmann Method (LBM) by analysing the algorithm and the parallel implementation thereof, to see what scale of fluid effects can be created with different computing resources. Broadly speaking, the simulation scale is measured by the detail of the final product, but this will be more rigorously defined in the testing section of the thesis.

The LBM is a cellular automata method 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. The rules are based on physical equations that predict the movement of fluid molecules that are moving and colliding.

There are many types of fluids and they can be classified into two groups: liquids or gases. The defining difference is that a liquid has a definite surface that can freely move, called the free-surface. This gives the portion of space that is occupied by the fluid. For the LBM, a Volume of Fluid (VOF) method uses the simulation of the underlying fluid interactions to track a free-surface.

In our context, computing resources range from a single Dual Core desktop to over a thousand computers in a cluster environment. These resources must be used as efficiently as possible and the LBM implementation must optimise the use of the resources by dividing the fluid simulation up between the cores or CPUs in the best possible way. This process is known as load balancing. Currently, there are only more straightforward parallel implementations that make use of static load balancing. In this case, the simulation is divided up once,

University of Cape Town

at the beginning. We implement this method of load balancing and a new dynamic method, that allows the shifting of the simulation division as time progresses.

These implementations are tested to calculate the length of time to simulate a scene with a given level of detail. In conjunction with tests of a single CPU LBM implementation, the efficiency of each of the algorithms can thus be measured. Once the efficiency of the implementations has been determined, recommendations can be made about the computing resources required to achieve a suitable level of detail within a given time frame. This is important for media producers, as they need fast turn around times during production and good estimates will tell us if the LBM is suitable for the media industry.

The next section provides a motivation for this thesis. The evaluation criteria and testing outline is in Section 1.2, while section 1.3 outlines the main contributions and results. The last section contains an overview of the thesis structure.

1.1 Motivation

Visual effects are widespread in today's media, particularly in films and advertisements. These effects enhance media and make them more entertaining for viewers. Some examples of scenes making use of visual effects include: sinking ships, exploding objects and non-existent characters created with computer graphics (CG). High quality visual effects are important, as they make the viewer feel immersed. Such effects are difficult to create by conventional means, either because the scenes required are not easily created in real life or because the costs are prohibitive. Sometimes the effects are added onto real camera footage, while there are many cases where entire scenes and even entire 3D animated films, such as *Shrek*, are created in CG. These effects can be created by artists, by using special camera techniques, or with computer software.

Fluids, such as water, are found in many scenes. They are the cause of some of the most impressive and complex natural effects seen in everyday life. Take the examples of a ship sinking or an object exploding. Media producers seek to recreate fluid effects, but scenes such as a city being swamped with water are not practically possible. It is very important to capture fluid behaviour correctly, as it is well defined by physical equations and viewers can easily tell when water is behaving incorrectly. In fully CG Media, all fluids have to be recreated, including honey, oil and air. The last is a particularly interesting case. Although it is not visible to the human eye, air can be treated as a fluid and its effects, such as wind, influence scene objects. Although the different types of fluid are related, the focus of this thesis will be on the class of fluids called liquids.

Simulation methods are a means of recreating real world behaviours without needing all the real world elements present. In the case of fluids, the individual molecules that make up the fluid are ignored. One means of simulating fluids is by creating down-scaled versions of the scene, for example, by filming a much smaller amount of water flooding a model of a city that is to be swamped. This gives a somewhat similar result to what would be seen for the full size city, but the different scale is often apparent in the final shot. These separately filmed shots can either be used as they are, or overlayed with other footage to provide the final shot, a process known as compositing. The fluid effects can also be drawn in two dimensions and overlayed onto the final image, but this requires an artist to draw the fluid for every frame. These two approaches do not use physical equations to govern the behaviour of fluids.

University of Cape Town

Fluid behaviour is caused by the molecular forces between millions, or even billions, of molecules making up the fluid. Due to the number of atoms and overall force complexity, simulating all the atoms and the forces is currently computationally not feasible [Stam, 1999]. Instead, approximate simulations are used to augment current scenes with visual fluid effects. The simulations often use low detail approximations or do not give the correct fluid behaviour due to the use of simplified physical equations [Foster and Metaxas, 1997]. Again, the fluid shots generated from the computer simulations are used as they are or are combined with other real world camera shots, with possible alteration by artists.

A more ideal approach is to fully simulate the fluid in 3D as closely as possible without input from the artist. This approach has become increasingly popular, as it provides the most realistic behaviour with the least input effort. This lowers the cost of producing a scene. Additionally, in a 3D scene it is not always possible for an artist to capture all fluid detail. Full computer simulations aim to recreate scenes with more detail than an artist would be able to achieve. Due to the faster processing power of today's Central Processing Units (CPUs) and improved simulation algorithms the full simulation approach has become more feasible [Irving et al., 2006].

Media producers have many deadlines to meet and the time to create a scene is crucial, hence the simulation times need to be as fast as possible. Individual CPUs have improved vastly in their performance and parallel chips with two and four cores are commonplace. One way to speed up the simulations is through the use of parallel algorithms, which make use of the extra cores on single CPUs and multiple CPUs, in environments such as clusters. The use of such techniques can bring simulations, which previously could only run on one CPU and took days or weeks, down to a matter of minutes or hours when running on hundreds of CPUs.

Naturally, the time required for the simulation will be relative to the amount of detail that is being simulated in the scene. For the LBM, which simulates fluids on a grid, the simulation scale is given by the simulation grid size [Thurey, 2007]. Effectively, the larger the number of grid cells, the more individual sections of fluid are being simulated. The more individual pieces of fluid simulated, the greater the quality of the final fluid simulation produced. Juxtaposed to this concept is the spatial resolution of the simulation, which measures how much each individual piece of fluid represents in the real world. One grid cell could correspond to $1m^3$ of fluid, where small droplets are not simulated, or $1cm^3$, where large bodies of water require large numbers of grid cells for accurate representation.

University of Cape Town

1.2 Aims and Evaluation

The target market is liquid animations for the film and animation industries. Both these markets require highly detailed fluid surfaces, but unlike fluids for a gaming platform, the simulation model need not run in real-time. However, to produce a good final product, an animator will need to examine the surface generated by the fluid model and may make modifications to the scene or the model to obtain acceptable results. This places restrictions on the running time of simulations, as a long run time reduces turn around time during production.

The aim of this thesis is to create a system that efficiently uses many CPUs to simulate fluid for a 3D scene within the target market's time frame. In particular, we investigate the scalability of the system with respect to required fluid detail. We have developed a system that is able to produce fluids by running physical simulation using the VOF LBM to produce fluid surfaces for a given 3D scene. We run scalability and efficiency tests, with respect to the number of CPUs used, using this implementation.

For the purposes of this thesis, three VOF LBM implementations will be created: for single CPU, for multi-CPU with static load balancing and a multi-CPU with dynamic load balancing.

The aims of this thesis are:

1. to produce large scale simulations of fluid;
2. To evaluate the robustness of the VOF LBM for simulating fluids with a free surface using a single CPU implementation;
3. to estimate the simulation scales and spatial resolution that can be created using a single CPU implementation of the VOF LBM;
4. to compare the performance and scalability, with respect to the number of CPUs, of the static and dynamic load balancing implementations of the VOF LBM;
5. to recommend the architecture required to achieve particular simulation scales using the parallel version of the VOF LBM.

Note, we developed two different implementations of the multi-CPU fluid simulation, one that makes use of the static load balancing and one with dynamic load balancing. Both will need to be evaluated with respect to the appropriate aims.

Evaluation of both the single - and multi-CPU implementations will consider whether the simulations produced are realistic, the possible range of values for the adjustable parameters and the performance (run time and memory consumption) of the system. An analysis of the effects that are achieved using this software will be performed. To our knowledge, a detailed analysis for the VOF LBM algorithm is not provided in literature. Firstly, it is important that the VOF LBM algorithm produces adequate fluid motion. As the target application is visual effects, the results do not need to be physically correct, but they must look realistic and have no visually jarring artifacts.

As with most simulations, there are a number of different parameters that are used in the algorithm. The different possible parameter values are tested to verify the robustness of the algorithm. We try to find out if there are points where the algorithm breaks and in which situations it can be used.

The performance of the single CPU implementation is also important. An analysis of the run time and memory consumption of the single CPU implementation is provided to determine at what scale it is feasible to produce fluid simulations with a single CPU.

The next part of the evaluation is concerned with the multi-CPU implementations; as these implementations may produce incorrect results, validation of the multi-CPU implementations is performed. We conduct a comparison of the same simulation generated by the single CPU implementation. A simulation is considered adequate as long as it looks realistic even if the results deviate slightly from the single CPU generated fluid.

To evaluate the performance and scalability of the two multi-CPU implementations, a number of different test scenes are constructed. These scenes are inspired by those currently required in today's visual effects. The

University of Cape Town

scenes are: a water fall scene, a wave breaking over a city (inspired from the movie *The Day After Tomorrow*[1]), Gnomon[2] showering in water/mud (inspired from *Shrek*[3]), Breaking Dam (from the literature [Thürey, 2003]) and water drop falling into a pool (also from literature [Pohl and Rüde, 2007]). The time taken for a complete simulation is compared against the number of CPUs used for the simulation. This provides an indication as to how efficient the VOF LBM is in a multi-CPU environment and at what level of fluid detail it is feasible to produce simulations.

1.3 Contributions and results

We provide a scalability analysis of the parallel VOF LBM simulation that makes use of a new dynamic load balancing scheme. No similar study can be found in literature.

In addition, several topics have been researched to contribute to this thesis. Firstly, the new scheme needs to be compared against the currently used static load balancing scheme and a scalability analysis of the parallel VOF LBM simulation using static load balancing is performed for this purpose.

As there is no easily accessible derivation and correctness proof for the LBM in current literature, simple versions are developed. A distributed fluid simulation plug-in for Houdini, which uses the VOF LBM for the physical simulation was created to allow scene specification. Houdini is a popular 3D software package created by SideFX[4]. It has been used in many of the latest movie titles, such as *Spiderman Three*, and has a single CPU fluid simulation tool, but no distributed version.

The fluid simulation will be used by artists and, therefore, the fluid animation should be produced correctly for different input parameters. Hence, an analysis of the single CPU VOF LBM implementation for robustness with respect to input parameters has been performed. With the this and the other analyses mentioned, we make feasibility recommendations for producing scenes of a specific detail.

The first result from this thesis is that the VOF LBM was found not to be very stable and requires extra stability enhancements. Due to these stability conditions, the length of a cube domain that can be simulated is of the order of $10cm$, which is far from ideal if a city flood is to be simulated. However, in practice, simulating a city at these orders of spatial resolution, produces reasonable final animations as shown in the results section.

For low grid resolutions, of the order of 100^3 (and hence low fluid detail), the parallel implementation of both the static and dynamic load balancing scale poorly. Higher resolutions, 300^3 and greater, scale at better rates. The dynamic load balancing only enhances the efficiency of simulations for lower numbers of CPUs and reduces the efficiency for large numbers of CPUs. In general, there is a crossover point where the static load balancing starts performing better then the dynamic version. The poor scaling at large CPU numbers for the dynamic simulation was found to be caused by badly chosen load balances. The maximum run time for a individual slave process for dynamic algorithm was worse than the static algorithm. This is due to local load balancing decisions.

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

University of Cape Town

The cause of the poor scaling was found to be the dependency of the parallel LBM algorithm on synchronisation. This requires constant communication, which remains the same for all numbers of CPUs. However, considering the amount of communication required, the scaling can be considered good.

The first recommendation made is that the general stability of the algorithm is poor and the basic stability implementations need to be replaced with more advanced measures. Secondly, we note that Infiniband is the best interconnect for use in the cluster environment. From the scalability analysis's, we conclude that the production a high resolution simulation at a grid size of 600^3 in a reasonable time frame requires a cluster of 200 CPUs. A reasonable time frame is defined relative to turn around time within a working day or week.

It is very important to establish simulation correctness and we recommend an implementation that can check for simulation correctness in an automated fashion. This is necessary when implementing optimisations, especially in a cluster environment, as the software can quickly become large and unmanageable.

The last recommendation is that the dynamic load balancing algorithm needs to use the global computation time for each slave process for better load balancing decisions. The current implementation, suffers from sub-optimal balances due to only using local information.

1.4 Thesis Structure

The thesis is comprised of seven chapters, which are organised as follows:

- Chapter 2 discusses computer simulations. It covers how simulations are defined and the general approaches to creating fluid using physical equations. The choice of the LBM is motivated.
- Chapter 3 then goes into the details of the physical equations from which the LBM is derived. This provides specifics required for the implementation of the method. The stability of the method is discussed.
- Chapter 4 provides details of the implementation of single CPU version for fluid simulation. The VOF surface representation is introduced and the integration of the simulation into the software package Houdini is outlined. The results of the simulation are provided to show what effects are achievable with the VOF LBM. The robustness of the method with respect to the input parameters is explored.
- The parallelization of the VOF LBM is presented in Chapter 5. Technologies for the parallelization are introduced and the problems arising from them are discussed. To improve the efficiency of the parallel implementation, we make use of load balancing. Both static and dynamic load balancing algorithms are outlined.
- Chapter 6 presents the results of the parallel simulations. A number of test cases are presented. The system correctness of the parallel implementation is validated against the single CPU implementation and the optimal choice of load balancing parameters are explored. Given these parameters, the scalability of the simulations is analysed. Further insight into the performance of the parallel version is gained from comparisons of the load balancing methods and code profiling.
- Lastly, the results are discussed and conclusions drawn in Chapter 7 and 8. Recommendations are given for the required or preferred architectures for producing fluid simulations at a given detail level.

University of Cape Town

Chapter 2

Background

There has been considerable research in the field of fluid simulation across many different disciplines, such as engineering, physics and computer graphics. The first two fields naturally seek methods that produce physically correct behaviour and the last, something that is visually convincing. This chapter introduces computer simulations and fluid concepts. Early approaches to fluid simulation are discussed, with an emphasis on the most popular methods and their application in computer graphics. In addition, we provide motivation for the choice of the Lattice Boltzmann Method for fluid simulation over other approaches and explain why the parallelization of fluid simulations is an interesting problem worth investigating. As advanced fluid effects are not within the scope of the thesis, such background information has been excluded from this chapter. The interested reader is referred to Appendix A for a summary of effects currently researched.

University of Cape Town

2.1 Computer Simulation

It is desirable to use computers to simulate the real world, as they can perform millions of calculations per second without supervision. However, to do this, real world objects need to be represented appropriately and computers only perform calculations to finite precision and store a limited amount of data. Thus, simulations may not be accurate and the simulation error must be considered to determine the validity of the simulation as a whole. The simulation error is the difference between the object representations and the actual objects in the real world. Once an object representation is chosen, a simulation algorithm is created that takes into account the chosen representation and computer limitations.

Most cases require a trade-off between the simulation error and the detail or speed of the simulation. For example, a 2D or 3D object can be represented as a triangle mesh. Figure 2.1 shows two possible representations for a 2D circle with different levels of detail. The first samples the space on a grid of step size $\Delta x = 1$ and the second on a grid with $\Delta x = 0.5$. The smaller step values allow representation of higher levels of details, with less error. However, at a higher level of detail, more information has to be stored and the trade-off is between detail and the memory required. In addition to the extra memory requirements, more triangles could result in more processing, if the complexity of the simulation algorithm is proportional to the number of triangles.

A simulation algorithm updates the object representations over time. The algorithm has rules, often based

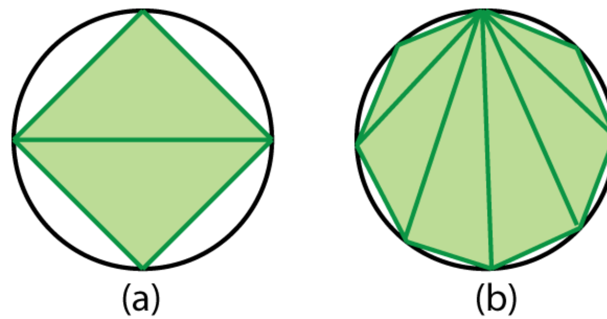


Figure 2.1: Two triangulations of a 2D sphere. (a) low resolution (b) high resolution. The triangulation on the left has a far larger error when used to represent the sphere than the right.

on physical laws, that describe how objects interact and their future positions. These are based on the relative positions of objects and the world state in the past and present times. To perform a perfect simulation, all the object positions have to be represented at every point in time. Of course, this is not possible and time must be discretised into time steps, Δt. The simulation then begins at time t_0 and updates the objects for the next time, $t_o + \Delta t$. The choice of Δt is constrained based on the target application (in this thesis we target visual media), the stability of the algorithm and the simulation error for a given time step. In film, the updated object state needs to change at least as fast as the frame rate. A typical frame rate is 25fps (frames per second), requiring a $\Delta t \leq \frac{1}{25}$.

The second constraint is far more restrictive. A stable algorithm is defined in Press et al. [1986] as an algorithm which does not significantly increase the error of the simulated objects with each algorithmic iteration. Consider the case where the objects in a very simple algorithm are a collection of floating point numbers. Each number has to be represented with a finite number of bits. Let the error in their representation be $O(\epsilon)$, with $\epsilon \ll 1$. Now, if after subsequent iterations, the error is the still $O(\epsilon)$, then the algorithm is called stable. If the error is $O(\epsilon^{-1})$, then the algorithm is called unstable.

Algorithmic stability can be based on a number of different factors. Often it requires Δt to be small. The result is that a simulation may require a number of iterations to update objects. For each time an object state is required by the application. Note that the stability condition may require that $\Delta t < \frac{1}{100}$, in which case, a large amount of information is being produced when a the frame is only needed every $\frac{1}{25}$th of a second. Thus, the algorithm may perform a large amount of extra work for the output generated. In general, there is a range of values for algorithm input parameters for which the algorithm remains stable. These values are not easily calculated and conservative choices are often necessary. The study of algorithm stability is a wide field [Higham, 2002] and is often very specific to a given algorithm, so there is no one rule that allows the choice of an optimal time or space step.

The third constraint is the size of the error in the physical simulation. Some simulations may be stable for a large range of time steps, but may still accumulate a high error and may require smaller time steps.

University of Cape Town

2.2 Fluid Simulation

This section introduces some of the common concepts when dealing with fluids and the behaviour that is required for people to believe that a simulation run on computer is actually real fluid.

A fluid is defined by Batchelor [2000] as a substance that, when pressure is applied, flows into a different shape. Flowing is the free movement of the substance over time. A fluid may offer some resistance, which is measured by the viscosity of the fluid. The higher the resistance it offers, the higher the viscosity. A solid can be regarded as fluid with infinite viscosity, as it does not flow at all when pressure is applied. Ideally, viscosity will be a parameter of the simulation and different fluids should be produced, based on this parameter. The unit of measurement for kinematic viscosity[1] is m^2s^{-1}. There are many ways to describe viscosity, but they are all equivalent. Effectively, kinematic viscosity is a measure of how quickly a fluid will accelerate due to movement of surrounding fluid.

There are several other important quantities when considering fluid behaviour. The first is mass, or how heavy the fluid is. Conservation of mass is one of the most important physical constraints. A fluid simulation should neither create or destroy fluid matter, except for cases where fluid is added to a scene, from a tap for example, or removed from a scene, at a sink hole. The mass should remain constant from one time step to the next, unless visual realism is the sole requirement, in which case the change in mass should be sufficiently small to be invisible.

Another quantity, already mentioned, is pressure, it is defined as force per unit volume. It describes the amount force a quantity of fluid at a given position exerts on the fluid around it. The pressure can be different for every point in the fluid and is denoted by the scalar field $P(x, y, z)$. When pressure is applied to a fluid from one side, it will slowly be transmitted to the other side of the fluid over time due to the intermolecular forces. Also important for fluids is density, defined as the mass per unit volume. This is not necessarily constant, but we will only be considering fluids with constant density for the purposes of this thesis. This family of fluids is known as incompressible fluids [Carlson, 2004]. Fluids such as water and oil are very close to being incompressible and it is mainly gases that are considered readily compressible. For the purpose of this thesis water and oil are treated as incompressible. Finally, the physical parameter velocity describes the movement of a body of fluid .

These quantities can also be described at different spatial levels or scales, which represent the amount of spatial detail being captured. The previous description of quantities is given at a macroscopic level, where only an aggregate behaviour of each of the underlying molecules is considered. At a microscopic scale the atomic interactions are important. For pressure, this would be an expression describing the number of collisions per second per unit area, as this would result in the corresponding forces per unit area on the macroscopic level. There is a third level, the mesoscopic level, which is in-between the macroscopic and microscopic levels. This considers a collection of molecules instead of just one. Some quantity representations may be more convenient at certain levels.

Fluids can be classified into two categories: Liquids and Gasses. Each of these categories has different physical characteristics. Most notably, a liquid has a free surface, which marks the presence or absence of fluid and adds additional physical constraints, while a gas has no clear boundary. However, the underlying fluid behaviour is the same, so the fluid simulation problem is split up into two parts: fluid velocity/pressure

[1] There are other definitions of viscosity, but this is the one most relevant to this thesis.

University of Cape Town

simulation and fluid surface representation.

Turbulence is one of the most important effects to consider for realistic fluid behaviour. This describes the chaotic behaviour of a fluid. An example of turbulent behaviour is a large sea wave breaking on a shore. The water churns and is far from but a constant flow of water. Turbulent flow is characterised as a part of the flow that has a highly differentiated and unpredictable pressure and velocity over a small space [Batchelor, 2000]. A good simulation should be able to produce turbulent behaviour at full detail. Vortices are also often associated with turbulence. These are spinning flows of a fluid, such as when water flows down a drain.

So far, the fluid has been described in a very physical sense. From a more everyday perspective, a fluid is something that splashes, infinitely divisible, sticks to objects, forms whirlpools and flows. Another interesting fluid phenomenon is bubbles, where air is trapped inside the fluid, or foam, as seen on sea waves. Foam occurs due to the surface tension of fluid. This is the attraction of the fluid surface to other fluid surfaces that causes a minimization of surface area [de Gennes et al., 2004]. The most common evidence of surface tension is the formation of spherical drops when water falls. These are all effects a simulation should support.

Finally, we consider how fluid interacts with its environment, such as rocks and trees. Furthermore, obstacles provide a direct method of controlling fluid movement. The way in which the water interacts with obstacles in the environment is defined in the boundary conditions. These conditions define how the fluid bounces or slips across an obstacle when they come in contact. It is possible to have very complex or simple boundary conditions and the resulting fluid simulations can be very different [Poinsot and Lele, 1992].

University of Cape Town

2.3 First Approaches and the Navier-Stokes Equations

There were various early approaches to modelling fluids. These can be differentiated into those which seek physical accuracy and those where the focus is on appearance. Many of the early approaches fall into the later category, as the computation power available was not sufficient for accurate simulations. In such cases, the physical properties of fluids are not considered (or not all the properties). Rather, the authors seek to create similar behaviour to what would be seen with the human eye. Our application is visual effects and this thesis focuses on similarly producing fluid that at least behaves as a person would expect. Of course, a physically correct simulation will also look correct to the human eye.

The smallest physical building block that causes fluid behaviour is the particle and a fluid is a collection of particles. Each particle exerts forces, both attraction and repulsion, on neighbouring particles. This is a simplistic view of how a fluid behaves, but is sufficient to reproduce complex movement. Miller and Pearce [1989] used this idea as a basis for fluid simulation. Particles at a close distance repel each other and those at a moderate distance attract. The particles can then be accelerated with the sum of all the forces. To use this approach, all particles have to be simulated and all forces calculated.

Fluids can be modelled in other simple ways. For example, Fournier and Reeves [1986] used parametric functions to model ocean waves. They modelled the waves as they move across different ocean floors and added particles to breaking waves, which take on the velocity given by the wave parametrization. This parametric function makes use, in part, of sinusoidal functions, which were used in other earlier approaches [Foster and Metaxas, 1996]. Sinusoidal functions are effective because they are wave functions, which are often as-

sociated with fluid movement. The method is fast, but does not allow for highly detailed interactions with the fluid and the look of the fluid is limited to the shape of the function, which is unsuitable for film applications, as the results are not realistic enough.

One of the more physically based approaches, applied by Kass and Miller [1990], makes use of shallow-water differential equations. These equations model the water in $2\frac{1}{2}$ dimensions -only a 2D plane of water is simulated. The water surface is approximated as a height-field. They also modelled rain and beach wave effects. Using these differential equations they produce a simple, stable method which only requires one simulation iteration per frame of animation [Kass and Miller, 1990]. The stability of this method is good and the equations are simple, low in computation cost and the animations fast to produce. However, the fluid velocity is only known at the surface, which is a problem for a full 3D scene with submerged objects. Also, object-fluid interactions are not taken into account and the model does not cater for full 3D fluid simulation.

Another study, by Chen and da Vitoria Lobo [1995], uses a 2D version of the Navier-Stokes equations. The Navier-Stokes equations are a set of partial differential equations that describe the change in macroscopic properties of a fluid over time. They are widely accepted in many disciplines (Computational Fluid Dynamics (CFD), physics and computer graphics) as accurate and are used as a standard for fluid simulation. As such, we will take a moment to consider these equations.

As mentioned, the important properties at any point in the fluid are the velocity and pressure. The change in these quantities is given by

$$\nabla \cdot \overrightarrow{u} = 0 \tag{2.1}$$

$$\frac{\partial \overrightarrow{u}}{\partial t} = -\overrightarrow{u}\nabla\overrightarrow{u} - \frac{1}{\rho}\nabla P + v\nabla^2\overrightarrow{u} + \overrightarrow{f} \tag{2.2}$$

where $\overrightarrow{u}(x, y, z)$ is the velocity at any point of the fluid, $P(x, y, z)$ the pressure, v the scalar viscosity and $\overrightarrow{f}(x, y, z)$ is a force term such as gravity. These equations describe a fluid's movement in three dimensions. The ∇ operator is known as the Jacobian and $\nabla \cdot \overrightarrow{u}$ is the divergence of the velocity[2]. Equation 2.1 is known as the zero divergence rule, since what flows into a point in space must flow out and fluid cannot be spontaneously created. This, together with the $-\frac{1}{\rho}\nabla P$ term controls the in-compressibility of the fluid. Many fluids are incompressible, such as water, oil and even gases in most normal physical situations [Foster and Metaxas, 1996].

The first term in Equation 2.2, $-\overrightarrow{u}\nabla\overrightarrow{u}$, is known as the advection term. Advection is movement of a quantity in accordance with the vector field [Bridson et al., 2006]. Here, the velocity is being advected by the velocity field, also called self advection. The third term, $v\nabla^2\overrightarrow{u}$, is called the diffusion term. Diffusion is the mixing of a quantity in space until the value is constant in the entire space. In this case, the velocity will continue to diffuse until it is equal everywhere. The Laplacian operator, ∇^2, gives a measure of how different the velocity is from the neighboring fluid velocities. The term, $\nabla^2\overrightarrow{u}$, is a vector that adjusts the velocity to the local average. A fluid with a high viscosity will seek to have a high level of agreement with the surrounding velocities and will tend quickly to the average local velocity.

A perfect numerical solution to these equations would be a great step towards obtaining exact fluid behaviour, but, due to the non-linear advection term, this is very computationally expensive and time consuming [Wei et al., 2004b]. There are a number of methods to solve these equations, which belong to one of three cat-

[2]See the notation section at the beginning of the thesis for a definition of the operators mentioned.

University of Cape Town

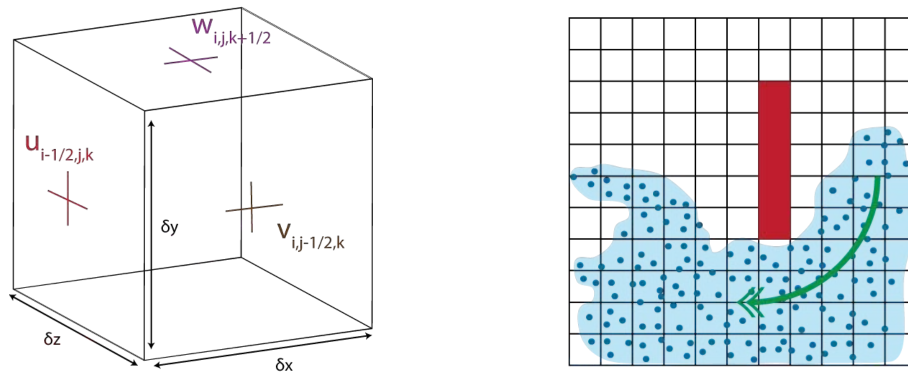


Figure 2.2: The Marker-And-Cell (MAC) grid and the markers used in the 2D MAC grid method. Left: The velocity components, stored logically, represent the corresponding fluid velocity component at the indicated positions on the cell, with the half index notation signifying the appropriate face. Right: The dark blue dots are the fluid particles that are advected at each time step, using velocities interpolated from the grid, while the light blue region is the fluid the particles are ideally simulating. The green arrow indicates the general motion of the fluid. Only cells containing marker particles are updated. The red cells indicate an obstacle.

egories: Eulerian (solving for pressure and velocity in discretised space), Lagrangian (particle based) and Semi-Lagrangian (a mixture of the two). In the subsequent section, we will look into advancements in these areas.

University of Cape Town

2.4 Simulation Methods

The evaluation of the Navier-Stokes equations provides the velocity and pressure for each point within a fluid. In Section 2.2, the fluid simulation problem is stated as having two parts, the evaluation of the velocity/pressure of the fluid and the surface representation. This section looks at how the first problem is solved. The following section will examine the second problem. In some sense, these problems are inherently linked, as the velocity and pressure of the fluid controls the movement of the surface.

The first of the methods for solving the Navier-Stokes equations, the Eulerian approach, asks the question, “What is the fluid’s velocity and pressure at a given point in space and how does that change from one time step to the next?” [Bridson et al., 2006]. To answer this question, the space which holds the fluid is discretized and the velocity and pressure are solved for on the discretized structure. The Lagrangian approach, instead of analysing each point in space, tracks the movement of particles, which carry the quantities with them, as they move. As there are advantages and disadvantages to both of the methods, they have been combined to form the semi-Lagrangian approach. This retains many similarities with the Eulaerian and has been grouped together in the same section below.

2.4.1 Eulerian and semi-Lagrangian Methods

2.4.1.1 Direct Navier-Stokes Solvers

Foster and Metaxas [1996] were one of the first to make use of 3D Navier-Stokes equations for computer graphics. They make use of previous research by Harlow and Welch [1965b], who use a finite difference

scheme coupled with a Marker-And-Cell (MAC) grid to evaluate the equations. A MAC grid stores the fluid velocities at the face centres for each face of a grid cell (see Figure 2.2). The pressure is stored at the center of the cell. For each time step, the following finite difference equation is used [Foster and Metaxas, 1996]:

$$\begin{aligned}
u^*_{i+1/2,j,k} \quad = \quad & u_{i+1/2,j,k} + \delta t \frac{1}{\delta x}[u^2_{i,j,k} - (u_{i+1,j,k})^2] \\
& + \frac{1}{\delta y}[(uv)_{i+1/2,j-1/2,k} - (uv)_{i+1/2,j+1/2,k}] \\
& + \frac{1}{\delta z}[(uw)_{i+1/2,j,k-1/2} - (uw)_{i+1/2,j,k+1/2}] \\
& + \frac{1}{\delta x}(\rho_{i,j,k} - \rho_{i+1,j,k}) \\
& + \frac{v}{\delta x^2}(u_{i+3/2,j,k} - 2 \times u_{i+1/2,j,k} + u_{i-1/2,j,k}) \\
& + \frac{v}{\delta y^2}(u_{i+1/2,j+1,k} - 2 \times u_{i+1/2,j,k} + u_{i+1/2,j-1,k}) \\
& + \frac{v}{\delta z^2}(u_{i+1/2,j,k+1} - 2 \times u_{i+1/2,j,k} + u_{i+1/2,j,k-1})
\end{aligned} \tag{2.3}$$

where u^* is the x-component of the velocity for the next time step. The symbols u, v, w are the x, y, z components, respectively, of velocity for the current time step and $\delta x, \delta y, \delta z$ are the grid resolutions along the respective axes. The equation seeks to solve the Navier-Stokes differential equation by stepping through space and time in finite increments. The $1/2$ indices denote one of the face positions. For the algorithm to remain stable $\max[u\frac{\delta t}{\delta x}, v\frac{\delta t}{\delta y}, w\frac{\delta t}{\delta z}] < 1$ [Foster and Metaxas, 1996]. For this method, an increase in the spatial resolutions requires an increase in time resolution of the algorithm and vice versa. The more detail required, the higher the computation cost. Using this given condition, to simulate fluid flowing at $1ms^{-1}$ at a scale of 1mm, $\delta x = 0.001m$, would require a time step of less then $0.001s$. Far more simulation steps would be required than animation frames generated.

After Equation 2.3 has been used to update the velocities, the velocity field may no longer have zero divergence. For a particular cell, this is corrected by using a similar finite difference form of Equation 2.1, which adjusts the pressure of a cell to counter the divergence created by the first step. Cell velocities are then adjusted according to the pressure. Adjusting one cell may affect a neighboring cell, so this is repeated for each cell. Foster and Metaxas [1996] conclude that their method has a high computational cost of $O(n^4)$, where n is the the number of cells along one axis of the grid.

As the previous method requires a large number of extra simulation steps to remain stable, Stam [1999] introduces an unconditionally stable semi-Lagrangian method. The method is unconditionally stable since any size time-step may be used and the simulation error remains bounded. His method splits up the solution of the Navier-Stokes equations into four steps (each step updates the velocity field). First the external force is used to accelerate the velocity field:

$$u_1(\vec{x}) = u_0(\vec{x}) + \triangle t f(\vec{x}).$$

Then the velocity field is updated by self advection. During a time-step, $\triangle t$, the velocity field will advect fluid around. The field at $\vec{x}(t)$ is updated by tracing a particle from $\vec{x}$ back through time, along the field lines, to its position at the previous time, $t - \triangle t$. The value of the velocity field at this position is used, namely

$$u_2(\vec{x}) = u_1(\vec{x}(t - \Delta t)).$$

University of Cape Town

Thirdly, the velocity is diffused by solving the linear system:

$$(I - v\triangle t\nabla^2)u_3(\vec{x}) = u_2(\vec{x}).$$

The system is solved for $u_3(\overrightarrow{x})$, using a partial differential equation solver. Lastly, the velocity field is corrected to make sure it has zero divergence. The *Helmholtz-Hodge Decomposition* states that any vector field can be decomposed into the sum of a divergence free vector field and a scalar field. Using this result, Stam [1999] projects the velocity onto a field with zero-divergence, by converting the projection into a sparse linear system, solvable in linear time. This is a global projection, as the entire velocity field must be known and projected.

The approach by Stam [1999] has become very popular and it, or its variations, are used by Fedkiw et al. [2001], Foster and Fedkiw [2001], Foster and Metaxas [1996], Carlson et al. [2004], Fattal and Lischinski [2004], Irving et al. [2006] and Losasso et al. [2006b]. However, due to the first order integration required, the approach suffers from numerical dissipation [Fedkiw et al., 2001], which causes fluids to be less turbulent then they are in reality.

To inject the energy lost to numerical dissipation into the field, Fedkiw et al. [2001] introduce "vorticity confinement" to computer graphics; the idea comes originally from CFD literature. They give the measure of the vorticity as the curl of the velocity field, $\omega = \nabla \cdot \overrightarrow{u}$, and add a forcing term to Equation 2.2 proportional to this measure. The authors claim that this method agrees with the full Navier-Stokes equations, but the magnitude of the forcing term has to be kept small for a stable simulation. A similar idea is used by Selle et al. [2005], but they use vortex particles to enhance the details of the fluid, where each particle carries a vorticity value. Particles are placed in the fluid flow at any point and use the velocity of the flow and the vorticity form of the Navier-stokes equations [3],

$$\omega_t + (\overrightarrow{u} \cdot \nabla)\omega - (\omega \cdot \nabla)\overrightarrow{u} = v\nabla^2\omega + \nabla \times f, \tag{2.4}$$

to evolve the particle positions and vorticity values, respectively, over time. Later, the vorticity at a point in the fluid is calculated by a weighted average of the nearby particles and energy is injected back into the fluid in a similar fashion to Fedkiw et al. [2001]. An application of these particles is given in Irving et al. [2006], where the vortex particles are placed behind a boat to add turbulence as it travels through water.

Currently, a 3D grid or voxel structure is required for an Eulerian simulation and the detail of the simulation is limited to the resolution of the grid. Some scenes may have different regions of detail, indeed, parts of the scene may be completely empty. Losasso et al. [2004] use an octree structure in place of the regular grid, with increased detail over certain regions. They show enhanced smoke simulation around a sphere, an object typically difficult to embed in a grid. Irving et al. [2006] use a height field structure for fluid below a surface, coupled with a full 3D grid on the surface. The structure is justified, as most detail will occur on the surface and the sub-surface level effects will be smoothed and less noticeable.

Departing from the traditional grid-based evaluations of the Navier-Stokes equations, many authors employ tetrahedral meshes. The grid based approach does not work well with irregularly shaped obstacles. Feldman et al. [2005a] improve the interface between a regular grid and objects by filling the gap with tetrahedral meshes. Similar to the design of the MAC grid, the velocity values are defined at the faces of the tetrahedron

[3] Obtained by taking the curl, $\nabla\times$, of Equation 2.2.

University of Cape Town

and the pressure at center. Klingner et al. [2006] generalises the stable method of Stam [1999] to tetrahedral meshes. As the obstacles are rarely static, Klingner et al. [2006] present a method for smoke animation that generates a new mesh for each iteration of the algorithm. The velocity values are transferred onto the new mesh using the semi-Lagrangian advection of Feldman et al. [2005b]. The pressure correction and forcing terms of the Navier-Stokes equations are then taken into account.

In conclusion, the direct Navier-Stokes solvers provide an unconditionally stable numerical scheme and allow arbitrary time steps to be used for the simulation. This has considerable advantages, as the speed of the simulation is not reduced by unneeded iterations. The most advanced and visually compelling simulations have been produced using the grid based solvers based on the method of Stam [1999].

The method, however, relies on a global pressure correction step to enforce incompressibility and this, unfortunately, increases the cost for conversion to a parallel architecture. Being a grid based approach, object interactions have to be translated into the grid structure, introducing possible boundary errors. Also, sub-grid effects, such as foam, cannot be modelled by the system.

2.4.1.2 Indirect Navier-Stokes Solvers using Cellular Automata methods

A cellular system is defined by Wolf-Gladrow [2000] as a “regular arrangement of single cells” with similar cell properties. These cells have states which change in discrete time steps. The rules that govern the change of state of a cell depend only on neighbouring cells. A cellular automata to describe fluid movement divides a region up into cells that cover the entire fluid and provide rules that update physical properties of each cell, such that the resulting physical behaviour will obey the Navier-Stokes equations.

The previous Eulerian approach simulated fluid from a macroscopic point of view. The alternative approach taken is to model the dynamics that produce the correct fluid behaviour. The fluid is now modelled at a mesoscopic level. We begin with the early methods in lattice cellular automata.

The first of the Lattice-gas cellular automata is called the HPP model (after the initials of the authors) and was introduced by Hardy et al. [1976]. This two dimensional method evenly divides up the region into squares, with a node placed at each corner (see Figure 2.3). Each node can hold four particles, which travel along one of four possible velocities to one of the neighbouring nodes. The particles, in effect, represent a collection of atoms. The model defines a collision operator and a streaming operator. The collision operator resolves the interaction of particles at a node, while the streaming operator propagates particles to neighbouring nodes.

The HPP model conserves mass and momentum [Kandhai et al., 1998], but fails to reproduce the macroscopic effects of the Navier-Stokes equations, due to the lack of rotational invariance [Wolf-Gladrow, 2000, Succi, 2001, Kandhai et al., 1998], as a rotation of the lattice yields different results for macroscopic values. Succi [2001] states that he also found that the method produced square vortices. Instead of the usual circular shape, the vortices have four flat edges that line up with the lattice.

Frisch et al. [1986] introduce the FHP model, which overcomes the lack of rotational invariance by dividing up the space into triangles and placing the nodes of the lattice on the corners of the triangles, giving the lattice hexagonal symmetry (see Figure 2.3), as each node is connected to six other nodes. Again, each node has six possible positions for particles, each of which can travel along one of the six possible velocity vectors. As

University of Cape Town

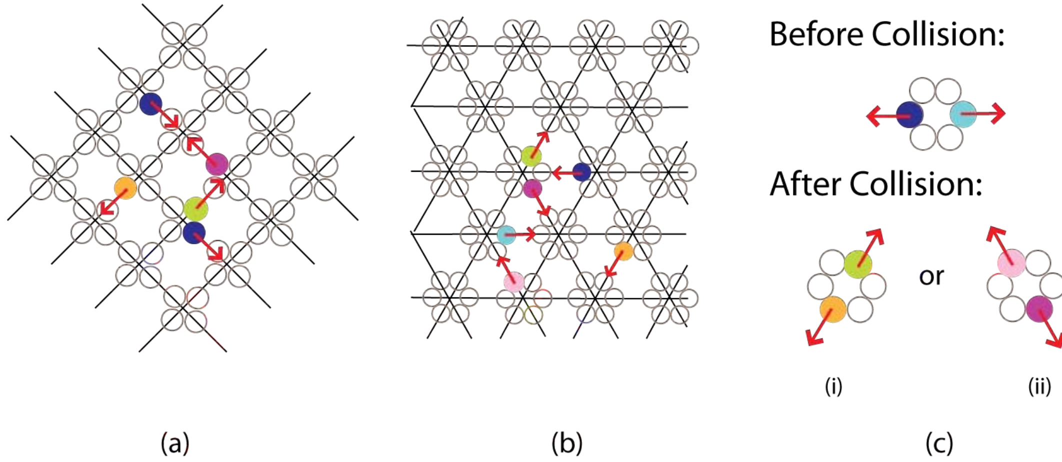


Figure 2.3: The HPP lattice
(a) The HPP lattice with four possible particle velocities, each particle associated with one velocity, given by a colour and the arrow indicating direction. The grey circles denote the possible particle positions, four per node at the corner of each square. A particle will travel one space to a neighbouring node. (b) The FHP lattice with six possible particle velocities. (c) An illustration of one of the boolean collision rules on the FHP lattice. The transition to state (i) or (ii) has a probability of 0.5.

with the HPP model, streaming and collisions rules are present. In this case, the collisions are given by

$$n_i(x + \vec{c_i}, t + 1) = n_i(x, t) + \omega_i(\vec{n}(x, t)) \qquad i = 1 \ldots 6, \tag{2.5}$$

where $\vec{c_i}$ is one of the six velocities, $n_i(x,t)$ a boolean variable indicating the presence of a particle at node x and velocity i, and ω_i is the collision operator, which outputs the influence on position i due to current state, given by $\vec{n}(x,t)$ (an example of this definition is shown in Figure 2.3). The collision operator is implemented as a set of Boolean equations based on the previous state.

This model is shown in Frisch et al. [1986] to be equivalent to the two dimensional Navier-Stokes equations, but only for low mach numbers (defined as the maximum velocity of the fluid over the speed of sound). The macroscopic values of density, ρ, momentum density, j, and velocity, u, are given by:

$$\begin{aligned} \rho(x,t) &:= \sum_i n_i(x,t), \\ \vec{j}(x,t) &:= \sum_i \vec{c_i} n_i(x,t) \qquad \textit{and} \\ \vec{u}(x,t) &= \frac{\vec{j}(x,t)}{\rho(x,t)} \end{aligned}$$

respectively. Both the HPP and FHP methods are applicable to massively parallel computing, due to the Single Instruction Multiple Data (SIMD) nature of the model. Another contribution of Frisch et al. [1986] is a proof that the equilibrium distribution of the particles for a given node can be predicted using the Fermi-Dirac distribution. The differential equations recovered by the FHP model are given by Kandhai et al. [1998] as

$$\frac{\partial v}{\partial t} = -\left(g(\rho)v \cdot \nabla\right) v - \frac{1}{\rho}\nabla p + v\nabla^2 v \tag{2.6}$$

University of Cape Town

These are similar to the Navier-Stokes equations, except for the $g(\rho) = \frac{\rho-3}{\rho-6}$ factor. However, for low mach numbers, time can be re-scaled, i.e. $t^* = \frac{t}{g(p)}$ [Wolf-Gladrow, 2000] and the full equations can be recovered with a scaling of the viscosity, $v^* = \frac{v}{g(\rho)}$. Hence, an adjustment to the required viscosity needs to be made. Other improvements to the model were made by introducing a rest particle for each node. This is known as the FHP II model [Frisch et al., 1987] and further modifications made to the collision operator in FHP III, allow the use of larger range of mach numbers. The drawback of the FHP methods is that the simulations produce noise artifacts and, to obtain pleasing results, the average over a large number of repetitions of the algorithm has to be used [McNamara and Zanetti, 1988]. This has a large computational requirement.

McNamara and Zanetti [1988] introduce the first of the Lattice-Boltzmann methods. They no longer use Boolean variables, n_i, for the presence of particles, but track the average value of the particle distributions, f_i, by using real numbers between 0 and 1. The collision operator is the same as the FHP model, but now substitutes the arithmetic operations "+" and "." for the Boolean operators "and" and "or" used by Frisch et al. [1987]. The use of particle distributions avoids statistical noise, hence the method is more efficient. As with the FHP model, the collision operator is a non-linear function of all possible particle positions at a node and is computationally expensive. To reduce the computational complexity, Higuera and Jiménez [1989] produce a quasi-linear collision operator. The operator gives the change of a given f_i as a linear combination of all the current node distribution functions. This advance is seen by Succi [2001] as the "most significant breakthrough in LBM theory" as simulations now become computationally practical.

The current linear combination of f_i, for all i, is derived from the rules in FHP, which state when a collision can occur between particles traveling along given velocities. These rules were chosen in such a way as to conserve momentum and mass and to give rise to the Navier-Stokes equations. Higuera et al. [1989] show that they are not the only rules that generate the equations and that other linear combinations still produce fluid behaviour. They effectively allow the simulation of flows with far lower viscosities than previous methods.

The collision operator was simplified even further by Qian et al. [1992], who use a relaxation technique in place of the operator. They use the equilibrium distribution,

$$f_i^e(t,x) = t_p \rho \left\{ 1 + \frac{c_i \cdot \overrightarrow{u}}{c_s^2} + \frac{u_\alpha u_\beta}{2c_s^2} \left(\frac{c_{i\alpha} c_{i\beta}}{c_s^2} - \delta_{\alpha\beta} \right) \right\},$$

to give f_i^e for a given velocity, $\overrightarrow{u}$. Here, c_s is the speed of sound, t_p is a weighting value to retain lattice symmetry (see paper for details) and α, β denotes the sum over Cartesian coordinates. Collisions are then resolved using the following:

$$f_i(x, t+1) = (1-\omega) f_i(x,t) + \omega f_i^e(x,t). \tag{2.7}$$

This operator only relies on one distribution function, the equilibrium distribution and the parameter ω. This is known as a relaxation method, as the current distribution, f_i, is relaxed towards equilibrium, f_e. The Lattice Boltzmann Method is based on this movement of particles from one cell to neighbouring cells and this collision operator.

The LBM was introduced into computer graphics by Thürey [2003], based on the use of LBM in metal foaming from Körner and Singer [2000] and Körner et al. [2002]. In these papers, the method is validated and tested against real fluids, such as a breaking dam experiment. Thürey [2003] is able to produce simulations of falling drops, breaking dams and bubbles rising in water. The stability of the simulations is not discussed extensively, but very small bounded time steps are required and stability problems are reported for low viscosities. The time

University of Cape Town

step in the lattice is limited by the fluid velocity as a fluid particle cannot travel more then one grid space per simulation iteration. If the step is too large collisions can no longer be resolved in a physically correct manner and the simulation becomes unstable.

With a fixed time step, the velocity of the fluid may become large and cause instabilities. Thürey et al. [2006] improve the stability of the simulation by introducing adaptive time steps. When the maximum velocity of the fluid is detected to be too large, the simulation is re-parametrized with respect to a new smaller time step. The new parametrization gives new lattice viscosities, such that the behaviour of the fluid at the new time step will agree with its previous behaviour and higher overall velocities can be simulated. Using this more stable algorithm, Thurey et al. [2005a] produce interactive free-surface simulations on grid sizes of 26^3 to 36^3, running at frame rates of 20-27 fps on a standard desktop PC.

To improve the performance of the LBM, Thürey and Rüde [2005] use an adaptive grid. Groups of LBM cells are aggregated in regions of low interest into one larger LBM cell. The interface between the large cell and smaller cells is modified to maintain correct behaviour. This approach reduces the running time of simulations by a third for the falling drop and breaking dam test cases. The use of larger grid sizes, however, is a new source for instabilities as the large grid can no longer resolve sub-grid turbulence. Thürey and Rüde [2008] use a sub-grid turbulence model to resolve the instabilities and produce high quality stable animations with approximately a 3.85 times improvement in speed.

In summary, the cellular automata methods provide a simple cellular automata style fluid representation, with equally simple update rules. During an update, only local information is needed and the method is inherently parallizable, as work units can be distributed to different locations (threads, processes or CPUs) and the cost of quantity synchronization will be low. This makes the possibility of parallel conversion straightforward. Complex boundary conditions are handled simply and quickly, as the particle distributions can individually interact with objects. Simple boundary conditions offer adequate fluid simulations, but there is room for improvement. Shankar and Sundar [2009] show that the solution obtained from the LBM simulation can be improved with more complex boundary conditions.

The LBM methods are applicable to computer graphics, but are very memory intensive, as they need to store a large amount of information for each grid cell. This is a drawback when modelling large scenes. The numerical scheme is not unconditionally stable for fluids with low velocities, so an adaptation needs to be used to introduce stability with a consequent added computation cost. Additionally, a maximum fluid velocity is imposed to ensure stability.

2.4.2 Lagrangian Methods

The Lagrangian methods use particles to represent the fluid. These particles carry all the fluid quantities with them as they travel through space. Mass is explicitly defined for each particle and mass is always conserved as the number of particles remains constant. Density is then related to the number of particles per unit volume, while pressure and velocity, for each particle, are then related to the surrounding particles. This is the general approach to handling Lagrangian fluids with particles, but some methods may have more specific or different approaches.

University of Cape Town

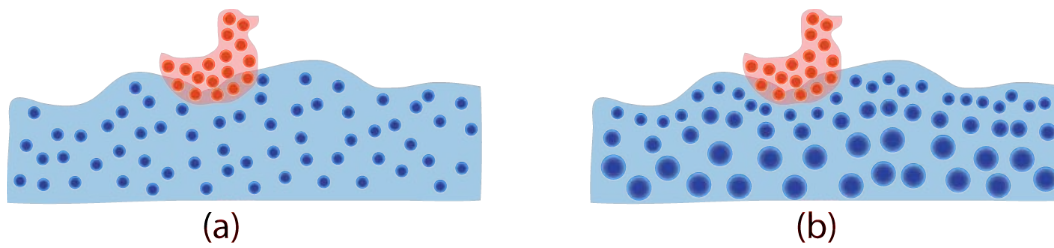


Figure 2.4: Fluid simulated using Smoothed Particle Hydrodynamics (SPH).
(a) The traditional SPH representation with uniform sized particles. (b) The multi-resolution SPH method with varying sized particles. Both show the rigid floating duck. A rigid object is populated with particles that form the basis for the object's interactions in the scene.

Smoothed Particle Hydrodynamics (SPH) treats the volume of a fluid as a collection of small regions and was first introduced to the graphics community for fluid simulation by Müller et al. [2003]. SPH embeds particles into a scene where fluid is present, and these particles are advected using the Navier-Stokes equations to calculate their acceleration, see Figure 2.4 (a). Each particle represents a small element of fluid, with mass m_j at position $\overrightarrow{r_j}$. The key to this method is the calculation of smoothed values of density and relative velocity over the collection of particles. The smoothed density field of a quantity, A_s, at $\overrightarrow{r}$ is given as the sum of all particle contributions, weighted by a kernel function, W, (such as a Gaussian distribution)

$$A_s(\overrightarrow{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\overrightarrow{r} - \overrightarrow{r_j}, h). \tag{2.8}$$

In the equation, A_j is the value of the quantity for particle j. The integral of the kernel function is normalized to 1, to ensure that no loss of mass occurs. The parameter h selects the influence radius of each particle; the further a particle is from another the less influence it has. An advantage to using particles is that mass is perfectly conserved and Equation 2.1 can be omitted, although this does introduce some compressibility effects. Furthermore, the advection term of Equation 2.2, $-u\nabla u$, can also be omitted as the particles are advected for each step of the simulation and this term is handled implicitly by the representation. With these simplifications, the acceleration of the particle i is

$$\overrightarrow{a_i} = \frac{d\overrightarrow{v_i}}{dt} = \frac{1}{\rho}\left(-\nabla P + \rho\overrightarrow{f} + \mu\nabla^2\overrightarrow{u}\right). \tag{2.9}$$

First the density for each particle is calculated using the smoothed density field equation, as this depends only on the mass and the kernel function. Then, the smoothed quantities P and $\overrightarrow{u}$ are calculated using Equation 2.8. An adjustment is needed to make the pressure and viscosity force symmetric between any two particles, making the method non-exact. Some details have been omitted and the interested reader is referred to the original paper. The author produces interactive simulations with frame rates of 25 fps and 5fps for water in a glass, but the surface of the water is “blobby” and unrealistic.

To improve previous results, the moving particle semi-implicit method (MPS) uses a very similar definition and algorithm to the SPH method, but includes an extra pressure correction step to ensure in-compressibility [Premoze et al., 2003]. This pressure correction mimics that of Stam [1999]. The MPS approach produces fluid simulations in a run time of about 3-4 minutes per frame of animation with 100 000-150 000 particles on a Pentium 4 1.7Ghz with 512Mb RAM. The final results look more realistic than Müller et al. [2003], but this is expected with a far higher particle count.

University of Cape Town

Keiser et al. [2006] introduce a multi-resolution particle approach, where a group of particles are represented as one particle, see Figure 2.4 (b). With this approach, they achieve a 6 times speedup, for some examples, over the single resolution method. However, for simulations of turbulent fluid, most of the fluid has to be represented at the finest resolution. For such cases, no matter which approach one uses, very fine data structures will be needed to properly simulate fluids, as the dynamics that produce the fluid movement only happen on a very small scale.

The authors also use a unified approach to modelling solid objects in the scene by sampling the objects with particles. The fluid particles then exert a force on each of the object particles and the combined forces then give the transformation of the rigid body. Rigid body-on-rigid body collisions are handled in the same way, by calculating the effect of particle collisions belonging to each body.

In conclusion, the Lagrangian methods conserve mass perfectly, and they are not confined to specific grid locations. The method is easily parallizable, as each of the particle updates are independent. Indeed, due the kernel function in the smoothed quantities, the influence can even be zero beyond a certain distance, making only locally close particles important during the update step. The particle approach also lends itself nicely to fluid-fluid interactions [Müller et al., 2005], deformable objects [Keiser et al., 2005] and representation of effects such as bubbles and foam [Losasso et al., 2008]. The particle methods can be coupled with other methods. Losasso et al. [2008] show that the popular Eulerian approach can be augmented with SPH particles to model foam and bubbles, which cannot be simulated on a grid.

The disadvantages of the particle methods are in the look and feel of the liquid. Once the particle positions have been simulated, the surface of the fluid must be generated. In general, it is hard to generate a smooth surface from particles and a large number of particles are required to make the simulation realistic [Thurey, 2007]. Simulations generated using the particle methods do not look as good as simulations using the Eulerian methods. Compressibility is, in general, allowed in these simulations, unless a costly pressure-correction step is used.

University of Cape Town

2.5 The Fluid Surface

There are many methods of representing the fluid surface. A common approach uses marker particles on the MAC grid [Harlow and Welch, 1965a, Foster and Metaxas, 1996], which show where the fluid exists in space (see Figure 2.2 (b)). The collection of all the particles is then used to represent the fluid surface. For each iteration, the particles are moved using the velocity of the fluid. Foster and Metaxas [1996] instead use a height-field to represent the surface of a liquid. Stam [1999] uses marker particles to indicate the occurrence of smoke in a scene, but use a combination of advected texture co-ordinates and density values to render the smoke.

Foster and Fedkiw [2001] define a particle level set, which uses particles to define the surface of a liquid. An implicit function is created, $\phi(x)$, which is defined by finding the closest particle to $\overrightarrow{x}$ and then subtracting a given radius, r, from $\sqrt{(\overrightarrow{x} - \overrightarrow{x_p})^2}$, where $\overrightarrow{x_p}$ is the center of the particle. The liquid surface is generated by sampling $\phi(x)$ on a grid and using the marching cubes algorithm to tessellate the $\phi(x) = 0$ iso-contour with

polygons. Both the particles and the level-set position are updated for each iteration. ϕ is updated using

$$\phi_t + u \cdot \nabla \phi = 0 \tag{2.10}$$

and the particles using the fluid velocity. Level sets are prone to volume loss [Foster and Metaxas, 1996, Foster and Fedkiw, 2001] due to the finite difference approximation needed to calculate the update given by Equation 2.10. To fix this, regions of $\phi(x)$ with large curvature, indicating a region of splashing, are built using the particle's position to alter the level-set. This gives a nice smooth surface for the whole liquid, in general, while regions of greater turbulence are represented with greater accuracy and effects such as splashing and foam can be modelled. The method only uses particles within the fluid.

The level set approach of Foster and Fedkiw [2001] focuses on modelling the liquid volume. Due to the problems with level sets, Enright et al. [2002] instead focus on the surface of the liquid to achieve more photo-realistic effects, which they regard as more important for application to movies. They build on work by Foster and Fedkiw [2001] using the same approach to develop the water's surface, but instead of using either the tessellated surface at $\phi = 0$ or the particle positions to represent the liquid surface, they use the particles as error correction term to the implicit function. In addition, particles are placed on the inside and outside of the fluid. This allows them to more correctly model phenomenon such as bubbles.

An alternative to the level set approach is the Volume of Fluid (VOF) method. Essentially, these methods track the fluid fill-fraction for each voxel in a grid. A fluid is given by a fraction of 1, while a gas is given by a fraction of 0. The advantage of VOF is that mass can be perfectly conserved, however, it is difficult to get and accurate curvature estimate [Bargteil et al., 2006]. The VOF method is used by Thürey [2003] in conjunction with the LBM method as the distribution functions of a lattice cell can be directly used to compute the exchange of fluid from one voxel to another.

Sussman [2003] combine the level set and VOF techniques to represent the fluid surface. Thin sheets of fluid can then be simulated and the volume preserved. Their method allows for higher accuracy, which in turn allows the use of lower grid resolutions, giving an increase in performance. However, the method still requires the VOF and level set calculations, which introduces significant overheads [Zhu and Bridson, 2005].

Lastly, in the case of Lagrangian simulations, a fluid surface can be built around particles in the simulation. This is done in a similar fashion to the level set implicit function, but for each frame of the animation. Müller et al. [2003] use point splatting and iso-contouring to create the fluid surface. Keiser et al. [2005] use simple surface particles called surfels that are small disk like objects oriented away from nearby associated particles within the SPH simulation. The surfels positions are updated using the associated particles movement. The fluid is rendered by standard point sampled object rendering techniques. Keiser et al. [2006] create a smooth dynamic surface using a Delaunay triangulation between the particles withing the SPH simulation and air particles placed on the outer edge of the fluid. The triangulation is simple and adapts to the resolution of the simulation.

University of Cape Town

2.6 Parallel Programming

This section introduces parallel simulations, parallel programming models, LBM parallelizations and technologies (Subsections 2.6.1 to 2.6.4, respectively). The first subsection provides some of the general problems and algorithm classifications for parallel simulations, while the second gives definitions for analysing and building programming models to predict program run times. The third looks at previous approaches used in conjunction with the LBM and the last describes some of the technologies that can be used for the parallel implementation of a simulation.

2.6.1 Parallel Simulations

As described in Section 2.1, a simulation consists of object representations that are updated for a number iterations based on other objects and state information. To parallelize a simulation, the algorithm must be well understood and the data dependencies between objects and state for each of the algorithm steps must be carefully considered. Parallel programs are limited by data dependencies, as they restrict the number of operations that can be executed concurrently. The more dependencies between objects, the more synchronisation required. Fujimoto [2001] classifies synchronisation into conservative and non-conservative approaches. A conservative approach ensures that all dependencies are met before performing an object update, while a non-conservative approach allows outdated data during an update step. Using outdated data may increase the error in the simulation. More synchronisations per iteration, in the conservative approach, will cause a slow down in the simulation, so a trade-off between simulation accuracy and speed is sometimes present.

It may be necessary to synchronise each process with every other process. This can be achieved through direct communication with all other processes or through a dependency chain. For example, a chain is created when process k needs to synchronise with process $k+1$ and process $k+1$ in turn must synchronise with process $k+2$ and so on. The result is that process k has to wait for the last process to finish its work before it can continue its computation, and one iteration of the simulation is as slow as the slowest process in the chain. To minimise the synchronisation times in such cases, the workload of all process must be kept as even as possible.

Different algorithms require different types of parallelization [Ramanathan et al., 2006]. The easiest to convert are the so called embarrassingly parallel algorithms. This class allows the update of simulation objects entirely independently of one another. The second class is coarse-grained applications, in which entire loops or functions are run on separate CPU's. Third is the fine grained class, in which low level operations are run in parallel using special operations. This class uses features such as streaming SIMD Extensions (SSE) vector instructions that can perform many floating point instructions for a single clock cycle.

2.6.2 Parallel Programming Models

A model of the program provides a good way of understanding problems that may be encountered and potential speed optimisations. Aoyama and Nakano [1999] provide a good model for analysis of any program being parallelized. Every program is made up of a part that is inherently sequential and a part that is parallelizable. The parallelizable portion is a section of code that can be split up to run on multiple CPUs. The split can follow the Single Program Multiple Data (SPMD) model or the Multiple Program Multiple Data (MPMD) model

University of Cape Town

according to Flynn's taxonomy [Flynn, 1972]. The first works well when there are many data items that can be updated independently and the second when different algorithms need to be executed on the same data. An example of an un-parallelizable program is one in which the update of each data item requires all data items and they can only be updated in sequence.

Once we have identified which portion of the program is parallelizable, we can begin to determine how well it will run on multiple CPUs. The total run time of a program is referred to as the wall clock time. This time can be expressed as $T_{TOT} = T_s + T_p$, where T_p is the portion of computation that can be parallelized and T_s is the remaining portion that has to run on one CPU. Now, if the program is run on N CPUs, in the ideal case, $T_{TOT} = T_s + \frac{T_p}{N}$[Amdahl, 1967]. Unfortunately, communication time, denoted by T_c, which is needed to synchronize the CPUs, adds to the overhead. Our final model is then

$$T_{TOT} = T_s + \frac{T_p}{N} + T_c.$$

This gives rise to Amdahl's law [Dongarra et al., 2003], which provides a lower bound on the run time for a program, namely $T_{TOT} \geqslant T_s$.

While the total run-time of the program is a good indication of the parallel performance, an analysis method is needed to determine how scalable and efficient the program is. System scalability is the measure of how much faster the program runs when using many CPUs. The first step for scalability studies is to ensure the single CPU implementation is running correctly and efficiently. To get a base line run time for scenes, the run time of the single CPU simulation is used, and this is compared to the run time on N CPUs. The standard definition [Dongarra et al., 2003] of speedup for the multi-CPU implementation is calculated as

$$\text{Speedup} = \frac{T_{single}}{T_N},$$

where T_{single} is the time for the single CPU implementation and T_N the time for N CPUs. T_{single} is different from T_1, which is the time taken for the multi-CPU implementation running on 1 CPU. A plot of speedup versus number of CPUs will show how well the implementation divides the workload. The definition of efficiency for the algorithm is calculated as

$$\text{Efficiency} = \frac{1}{N} \times \text{Speedup} \tag{2.11}$$

and indicates how well the simulation divides over N CPUs. A perfect distribution would give an efficiency measure of 1.

The communication time can be refined further, so that $T_c = T_{start-up} + nT_{data}$. The first portion is the latency time, $T_{start-up}$, required for handshakes and buffer set up. The buffer is required to optimise transfer time by storing data and sending it when the network becomes available. This way the program does not have to wait for a slow network connection and can continue processing asynchronously while the send happens. This process is also used to overlap computation and communication. Finally, the time taken to send the buffer is given by nT_{data}, where n is the size of the data in bytes and T_{data} is the time to send one byte of data. When designing message sends, one can optimise them to take into account these values, but results may vary for different architectures.

Amdahl's law does not hold in some circumstances and superlinear speedup is achievable. A superlinear speedup occurs when, for N CPUs, the speedup achieved is greater then N. Beane [2004] shows the cases

University of Cape Town

where Amdahl's law does not hold are due to multiple levels of cache, most notably the "on-chip" cache. As more CPUs are used, each with their own cache, it is possible for the entire data and program to fit into L1 cache and the cache misses that occur with lower numbers of CPUs are circumvented. The author gives four properties that an algorithm must have to improve the probability of achieving superlinear speed-up. The first is that the algorithm must iterate many times over a large data-set. Secondly, the parallel workload for each CPU must diminish proportionally with the number of CPUs. Thirdly, $T_s + T_c$ must be low and, lastly, super linear speedup is more likely if the algorithm uses out of order memory accesses. Superlinear speed up is demonstrated by Venkatesh et al. [2004], who report a speedup of 11.05 for parallel CFD code on 8 processes, a 37.5% super linear speedup.

2.6.3 LBM Parallelizations

A multi-CPU algorithm should evenly balance the processing load and assign portions of work to each available processor, weighted according to their processing power, while minimizing the communication cost. Various load balancing methods have been designed with this goal in mind. Some methods are static, where the load is only balanced at the start of the simulation, while others are dynamic and the load balance changes during the run time of the simulation. Fluid motion introduces additional difficulties and the load balancing scheme must be able to change work division dynamically to account for the fluid's positional changes.

For LBM methods without free surfaces, a popular method for work assignment is domain decomposition, in the form of the slice, box and cube methods [Desplat et al., 2001, Amati et al., 1997, Pohl and Rüde, 2007]. As their names suggest, these work by dividing the domain according to the respective geometric shape. These methods are, however, general frameworks and can be implemented statically or dynamically. Kandhai et al. [1998] introduce a new method called Orthogonal Recursive Bisection (ORB). They compare this method, which takes into account the geometry of the scene by splitting the domain recursively with orthogonal splitting planes, to the box and slice method, which does not consider scene geometry. The results are very poor for the box and slice method, but this is expected given the authors unoptimized implementation of the algorithms, and very good for their ORB method.

Desplat et al. [2001]'s regular domain decomposition divides up the space into equal volumes of unspecified shape. Objects are moved within the domain, with the assumption that they will be evenly distributed and the decomposition remains static. More complex methods of decomposition are required for uneven distributions of fluid. Berger and Bokhari [1987] introduce the Recursive Bisection Method (RCB). They choose the axis along which computation (in our case fluid) is most widely spread and divide the domain orthogonally on that axis so that the workload is divided in half. They repeat this division for each sub-domain, until the domains are sufficiently small.

The LBM grid can be seen as a highly structured graph. There is considerable research into decomposing an arbitrarily connected graph into equal partitions, while minimising the edges between each of the partitions. This problem is known to be NP-hard [Karypis and Kumar, 1999]. They use a sophisticated method of multi-level partitioning, where the graph is first partitioned at a coarse level (generated by merging vertices and removing less important edges) then the partition is refined by including a finer level of detail (including some of the vertices or edges that were previously removed from the graph). However, due to the highly structured nature of the LBM grid, simpler methods will suffice for our purposes.

University of Cape Town

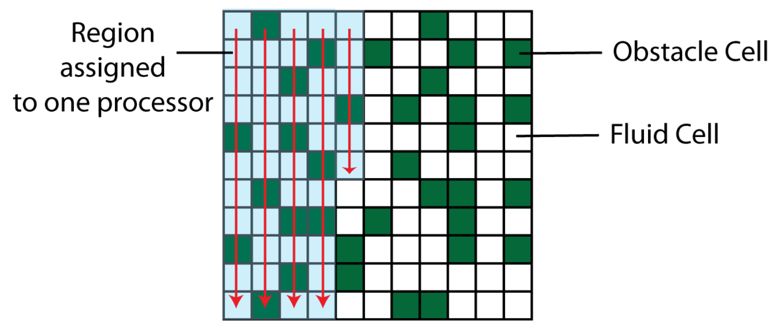


Figure 2.5: Irregular domain decomposition.
For a 2D lattice, the assignment of LBM grid cells to processes begins with one corner of the domain and continues down a column, proceeding until the required number of cells is reached. This creates irregular, jagged boundaries that are tracked by an adjacency matrix.

Wang et al. [2005] cater for complex boundary conditions by assigning N/P fluid cells to each processor, where N is the total number of fluid cells and P the number of processors. The fluid cells are chosen such that they are as close as possible to each other (Figure 2.5). However, it is not always possible to have flat interfaces between the domain divisions, which introduces higher communication costs. This method works well for static complex geometries, such as porous materials, because the method counts the fluid cells and automatically adjusts the domain sizes. The final domains are not equal in size, but are equal with respect to fluid contained.

Fluid with a free surface is inherently less balanced, particularly when in motion. To solve this problem, Körner et al. [2006] suggest using slice domain decomposition and tracking the fluid's movement over time. The initial decomposition is made by dividing up the volume of the fluid equally among all processors. A processor, numbered P, will then transmit its upper row of cells to $P+1$, and $P+1$ its lower row of cells to P[4]. If P finds it is waiting too long for the data to arrive, it will enlarge its domain by one row of cells by ordering $P+1$ to reduce its domain. The workload will balance over time. The authors point out that, because the fluid only moves a maximum of 1 cell space per time step, this form of load balancing will work well. However, they do not implement their ideas and the method remains untested.

University of Cape Town

In a similar fashion, Pohl and Rüde [2007] use slice domain decomposition, but rely on gravity to load balance their simulation and use a static slice width. Gravity causes the fluid to eventually gather at the bottom of the domain and the slices are divided up along an axis parallel to gravity, so the simulation ultimately achieves a good load balance. This holds true for scenes with a flat floor, but some scenes may have irregular flooring, such as river with varying depth. Another case in which a gravity approach is not optimal is when a scene is gradually filling up from one side. Here, the fluid may take a long time to fill the scene and there might even be an obstacle that completely blocks the flow of the fluid to parts of the scene, in which case some CPUs may remain completely idle.

[4] Unless it is the first or last processor, in which case one will be left out.

2.6.4 Technologies

A popular technology for the parallelization of a simulation is the Message Passing Interface (MPI) communication protocol, which is specifically designed to aid programming of parallel systems. Essentially, MPI defines functions that allow the programmer to pass data from one process to another in a similar fashion to sockets, but simplifications have been made to hide the usual network specifics, and management logic has been added to control a fleet of processes. These may be run concurrently on the same or a number of different CPUs. The key focus of the protocol is program scalability, allowing a MPI-based program to run on from one to thousands of CPUs simultaneously and MPI focuses on portability/generality.

An alternative to MPI is OpenMP, which is a thread based parallel programming technique for shared memory systems as opposed to the process based approach of MPI for distributed memory architectures. MPI, however, allows the use of shared memory (when available, to improve performance) instead of socket communication. Krawezik [2003] shows that OpenMP does not offer significant performance increases over MPI on shared memory systems. He compares MPI implementations of the NAS Benchmarks [Bailey et al., 1991] against various OpenMP implementations and finds that OpenMP does offer some improvements, but they do not necessarily warrant the use of the programming environment.

2.7 Using High Performance Computing

Our goal is to produce a system that is capable of producing highly detailed fluid in complex scenes. Fluid simulations have two potential bottlenecks: the number of floating point operations (due to the use of physical equations) and memory access (due to the large amount of data that needs to be simulated for detailed scenes). Both of these bottlenecks become exaggerated when one wants to produce a large scene with high quality. The time needed to run such a simulation on a single CPU is not practical. One may wish to distribute the simulation across a cluster or any computing resources available; this could be a number of clusters or possibly even grid computing resource.

Special architectures have been developed to deal with the algorithms using of many floating point operations and high memory bandwidth. One such technology is the Cell broadband engine architecture (CBE), which is a multi-core CPU. An example of an application is given in Sturmer et al. [2007]. The authors use this architecture, running a LBM fluid simulation, to simulate 2 heartbeats in 11 minutes. They achieve up to 4 times speed up over the same simulation software on a quad core XEON system running at 3Ghz.

Another attractive approach taken by a number of authors is the use of Graphics Processing Unit (GPU) acceleration. GPUs allow a very high degree of parallelism and can perform many floating point operations at the same time. The newest graphics card from NVIDIA, the GTX 280, has a theoretical maximum performance of 933GFlops. Wei et al. [2003] use a LBM on the GPU to simulate wind effects on objects such as feathers and bubbles. The use of the GPU gives a 9 times speed up over their CPU implementation. Li et al. [2003] also implement LBM on the GPU with a speed up of 50, but comment on the reduced accuracy due to the single precision nature of GPUs. They produce animations of steam and smoke. Keenan Crane produced real-time fluid simulations at grid resolutions of 64x64x128 on the GPU using the popular stable fluids method (see Section 2.4.1.1) of Howes and Thomas [2007]. While there has been progress made and Fan et al. [2004] have shown real time simulation of the underlying fluid dynamics, at the time of writing there is no known solution

University of Cape Town

for realtime free surface flows on a GPU [Thuerey, 2008].

In feature films such as *Poseidon*, the fluid simulations are distributed across a group of 5 Networked Linux PCs and run using an in-house software package, known as RealFlow [5]. Unfortunately they do not report the time taken for the simulations. On their benchmarking page, they report a simulation time of 30 mins for a simple breaking dam example on an AMD Opteron machine with 32 processors. This time frame is acceptable for use in a production environment, but this is a simple scene. During the making of *Surf's Up,* the breaking waves were simulated overnight on a cluster to be available for animators the next morning. Due to the unpredictability of the simulations and the length of time for a single run, multiple initial conditions were run, so that the best one could be chosen the next morning [Carlson, 2007].

There has been wide adoption of the use of multi-core programming to speed up simulations. In an interview, Ron Fedkiw, of Industrial Light + Magic, said that the ability to adapt algorithms for simulations to multi-core environments has been key to the high quality simulations we see in current movies. He also added that now scene changes by the animators can be integrated quickly into the fluid simulations [Peszko, 2006]. In the film *300,* a full battle scene on the ocean, where objects influence the water and not vice versa, is simulated using a cluster of about 80 PCs [Trojansky et al., 2007]. This allows for a very large scene as there is the combined memory and processing power of all the PCs. However, the programming overhead is far higher and the network backbone becomes important. These issues will be further discussed in Chapter 5.

University of Cape Town

2.8 Conclusion

When considering a simulation system, one can mix and match many of the methods described here in any number of combinations. However, some may be more suitable for certain situations. Furthermore, certain surface representations may combine better with a given dynamics simulation. It is important to choose the correct methods for a given application. In our case this is large scenes with high detail and dynamic objects for movies. To this end we choose the Lattice Boltzmann Method.

This uses a simple representation that handles dynamic boundary conditions well. The simplicity of the algorithm is important, as an efficient implementation will have fewer lines of code, be easier to debug and allows for more straightforward extensions. The VOF method applies well to the LBM, as movement of fluid mass is directly equal to the distribution functions, and will be used in our simulations as the chosen fluid surface representation.

The LBM is not very well known in the graphics community, but is a promising avenue of research. Geller et al. [2006] give a comparison of the efficiency comparison of the LBM and the traditional Finite-Element method (a generalisation of the finite difference approach with similar characteristics). They conclude that the LBM is “at least as competitive” as the traditional method, but only for more turbulent cases. For films, in general, we are only interested in turbulent cases, as the calm fluid cases are easily created by hand.

There are many approximations that reduce the computation time of fluid simulations, but these are mostly at the cost of lower scene detail and less turbulence. We seek to preserve as much detail as possible. As such, we will aim to make used of high resolution grids. In order to make the simulation times practical, use of

[5] www.RealFlow.com

HPC technology will be required. Parallel computing resources, such as a cluster, are used, as this gives us access to larger memory areas and high computation rates. Use of the GPU and CBE are also plausible, but due to the high degree of branching required for the VOF representation, might not perform as well. The LBM is particularly suited to such parallel environments as only local information (neighbouring cells) is needed during the update of a cell.

We investigate a LBM VOF implementation on a cluster architecture. An efficient solution is an interesting problem to solve as parallel fluid simulation has become increasingly popular with the price of hardware per GHz continuing to drop over recent years. More and more animation companies own their own clusters to use for computation tasks and have access to other similar resources. There are many centers around the world and proper use of the resources will allow us to produce high quality simulations. For instance, the Centre for High Performance Computing in South Africa gives access to a cluster of 160 nodes, each with the two dual core XEON's and 17Gb memory.

University of Cape Town

Chapter 3

LBM Fluid Simulation

University of Cape Town

One of our purposes in this thesis is to build a system that produces fluid for a 3D scene by modelling its physical characteristics. This chapter builds on information presented in Chapter 2 and describes, with more detail, the physical quantities used to model the fluid and the underlying equations. It avoids a more formal description of the data structures used in the implementation and the details about the simulation of the fluid surface. For that, the reader is referred to the subsequent chapter.

The chapter begins by providing a basic description of the Lattice Boltzmann Method (LBM) in 3D and the steps required to use it to simulate fluid in Section 3.1. The Boltzmann equation is then introduced in Section 3.2. This is the fundamental equation that drives the fluid behavior and from this the LBM is derived. From the model, Section 3.2.2 then describes important details taken from the proof that the Navier-Stokes equations (see Section 2.3) can be recovered. As with all numerical simulations, the numerical stability is important and requires discussion (Section 3.2.3).

3.1 Full Algorithm Description for 3D

To add fluid to a scene using the LBM, the scene is first divided up into a 3D grid. Each grid point or cell stores information about velocity, density and particle distribution functions of the fluid at a point in space. For the moment, we assume that there is no free surface, i.e. no division between fluid and air, and the whole scene is completely filled with fluid. The particle distribution functions seek to simulate atomic interactions such as attraction and repulsions that occur within the fluid. These interactions are modelled with cellular automata rules that use physical equations to update grid cells. The rules describe how the particles move within the fluid, which is termed streaming, and also how they collide with one another.

A LBM grid is called a lattice, as it is a regular arrangement of squares (2D) or cuboids (3D). Figure 3.1 gives the general structure of the D3D19 lattice. The name of the lattice comes from the 19 distribution functions (D19) stored for each grid position with each distribution function assigned to a particular velocity in three dimensions (D3). There are many popular variants of lattices for different dimensions: D2D9, D3D15 and D3D27 [Wei et al., 2004a]. Naturally, the simpler the lattice the faster the simulation.

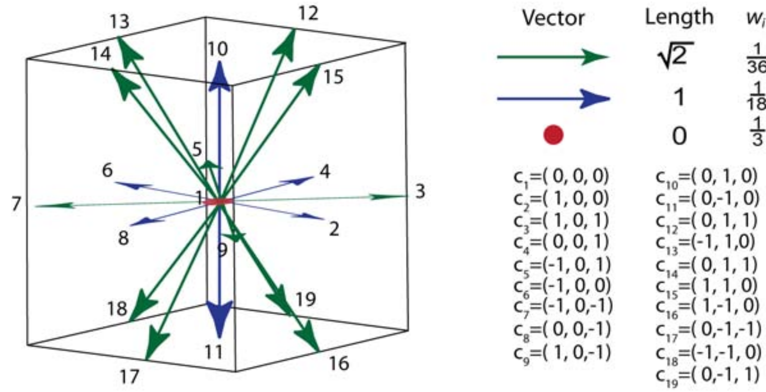


Figure 3.1: The 3D lattice that we use is commonly known as the D3D19 lattice.
The lattice has 19 different lattice velocities. Each vector represents a possible velocity along which a distribution function in the fluid can interact. These vectors allow distributions belonging to the current cell to travel to neighbouring cells. The c_1 vector gives the rest distribution function. The weights, w_i, used in the equilibrium function are given on the right.

We will use $f_i(x, y, z, t) = f_i(\overrightarrow{x}, t)$ to denote the distribution function along velocity $\overrightarrow{c_i}$ at grid location (x, y, z). With respect to notation, we will use f_i and $f_i(\overrightarrow{x}, t)$ interchangeably. The distribution function f_i can be viewed in two ways: (1) as the probability of finding a particle along velocity $\overrightarrow{c_i}$, or (2) as the measure of the fluid at a grid location distributed along velocity $\overrightarrow{c_i}$.

The simulation runs in two basic steps:

1. Stream
2. 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 need to remain consistent during the copy operation. The stream step can be summarised as

$$f_i(\overrightarrow{x} + \overrightarrow{c_i}, t + 1) = f_i(\overrightarrow{x}, t) \tag{3.1}$$

Once the particles have moved to a new grid location, the collisions between the new collection of distribution functions must be simulated. An example of a collision could be a group of particles, f_k, moving along some none zero velocity that hit a group of rest particles, 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. The collisions are resolved by relaxing each f_i towards an equilibrium distribution, which describes the ideal distribution of the particles along each $\overrightarrow{c_i}$ for a given fluid velocity, $\overrightarrow{u}(x, y, z)$, and density, $\rho(x, y, z)$, at a point. It is given by the following equation:

$$f_i^e = \rho w_i[1 + 3\overrightarrow{u}\overrightarrow{c_i} - \frac{3(\overrightarrow{u})^2}{2} + \frac{9(\overrightarrow{u}\overrightarrow{c_i})^2}{2}]. \tag{3.2}$$

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

University of Cape Town

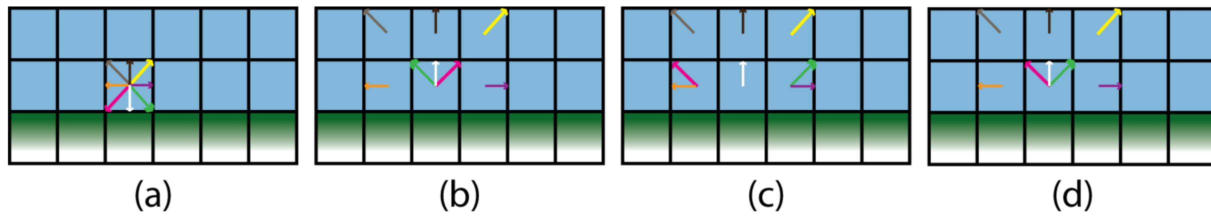


Figure 3.2: Distribution functions and obstacles.
During the stream step, if a distribution function would collide with a obstacle, it is instead reflected in a few possible ways. (a) the distribution functions before the stream step, (b) the distribution functions after streaming using the No-Slip boundary conditions, (c) using the free-slip conditions where the distribution function is reflected about the obstacle normal and copied to a neighbouring cell and (d) a hybrid which reflects the distribution function, but still retains it locally. The diagram indicates the difference between (c) and (d) with the pink and green arrows swapped.

most closely matches the fluid velocity. The velocity and density are defined as

$$\sum_i f_i = \rho, \text{ and } \sum_i f_i \vec{c_i} = \rho \vec{u}. \tag{3.3}$$

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

$$f_i(\vec{x}, t+1) = (1 - \frac{1}{\tau}) f_i(\vec{x}, t) + \frac{1}{\tau} f_i^e(\rho, \vec{u}). \tag{3.4}$$

The larger a value for 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 (see section 3.2.3).

3.1.1 Obstacles

Every 3D scene requires obstacles and a mechanism is needed to incorporate objects into the representation of the scene. Static objects can be added to any fluid scene by marking grid cells as occupied by the object. Continuing with the particle nature of the simulation, the stream step is then modified to account for the new type of cell. With a perfectly slippery surface particles will bounce off an object with their velocity reflected about the normal of the object surface at the point of impact. However, a surface may have friction and, in this case, the velocity should not be purely reflected. The two simplest boundary conditions are known as the no-slip and free-slip conditions [Thurey, 2007, Succi, 2001]. Figure 3.2 shows the difference. No-slip boundary conditions (Figure 3.2(b)) results in both a zero normal velocity and zero tangential velocity as the distribution functions are reflected about the normal and the tangent vector. The no-slip condition changes Equation 3.1, when $\vec{x}$ is an obstacle and f_i would be streamed, from $\vec{x}$, into

$$f_i(\vec{x} + \vec{c_i}, t+1) = f_{\tilde{i}}(\vec{x} + \vec{c_i}, t). \tag{3.5}$$

Above, $\tilde{i}$ denotes the inverse index of i such that $\vec{c_{\tilde{i}}} = -\vec{c_i}$.

Free-slip conditions (Figure 3.2(c)) reflect the distribution functions around the normal only, resulting in a zero

University of Cape Town

normal velocity, but leaving the tangential velocity as is. This gives

$$f_i(\overrightarrow{x} + \overrightarrow{c_i}, t+1) = f_r(\overrightarrow{x} + \overrightarrow{c_{FS}}, t).$$

Here, r is a new index, which gives the reflection vector of i and *FS* is a value to choose which neighbouring cell the distribution function should be fetched from. The disadvantage of free-slip conditions is that when an obstacle is found during streaming, extra neighbourhood look-ups are necessary. There are some tricky cases in corners when it is not obvious where the distribution function should be fetched from and a neighbourhood search must be performed. This complicates the code and reduces efficiency, as the condition is no longer a simple look-up table. It is also possible to combine the two previous conditions, so that, depending on the obstacle's material, more or less slipping can occur [Thurey, 2007].

We use a simple hybrid of the two, shown as (d) in figure 3.2. Here, a colliding distribution function is reflected about the normal only and retained locally in the current cell. This allows the correct particle bounce-back behaviour and the tangential velocity is no longer zero as with the no-slip conditions. Further, this all occurs locally, so no extra neighbour look-ups are necessary. We now have

$$f_i(\overrightarrow{x} + \overrightarrow{c_i}, t+1) = f_r(\overrightarrow{x} + \overrightarrow{c_i}, t). \tag{3.6}$$

3.1.2 Gravity and external forces

Until now, gravity has been excluded from the model, but it is easily added during the collide step after the calculation of the fluid velocity, by accelerating the velocity with the gravity constant. This is possible due to two actions that are performed in the collide step. The first is the calculation of the aggregate cell velocity from the distribution functions. This velocity is stored and used in the calculation of the new distribution functions for the next time step. However, the velocity can be freely manipulated before the second action. If, for instance, there was some reason to zero the velocity for some points in the grid, this would be possible here. Instead, the natural action of adding the acceleration due to gravity is performed. Effectively, the new distribution functions calculated are distributed towards the side of the cell that matches the gravity vector direction.

At this point, it is also possible to add control forces generated from an external source. Thürey et al. [2006] use control particles with attraction forces, to pull fluid towards certain positions, and velocity forces, to make the fluid match certain velocities. This is performed on multiple scales to preserve small fluid details while allowing control of overall fluid behaviour.

3.1.3 Simulation Initialisation

The model described so far has assumed that most values, such as the time step and space step, are correctly normalised. In the real world the scene division will not be as straightforward. Hence, a translation is needed from the scene dimensions into the lattice dimensions. Following Thurey et al. [2005a], the initial values for the lattice are outlined here. The first parameter chosen is Δx, the length of a grid cell. Since the lattice is a cube, there is only one length parameter. Once this has been chosen, the time step is selected to fulfill the stability

University of Cape Town

conditions (see section 3.2.3), expressed as

$$\Delta t = \sqrt{\frac{g_c \Delta x}{|g|}}, \tag{3.7}$$

where g is gravity and $g_c = 0.005$ is a constant chosen to limit the time step to a small value to maintain stability. The kinematic viscosity, given in $m^2 s^{-1}$, of the fluid is normalised to lattice units by

$$v_{lattice} = v_{fluid} \frac{\Delta t}{(\Delta x)^2}. \tag{3.8}$$

Gravity is also converted into lattice units as

$$g_{lattice} = g_{fluid} \frac{(\Delta t)^2}{\Delta x}. \tag{3.9}$$

Lastly, the relaxation parameter τ is calculated as

$$\tau = 3v_{lattice} + \frac{1}{2} \tag{3.10}$$

The individual distribution functions for each grid location are initialised using the equilibrium distribution with a given velocity, usually zero, in which case the distribution functions are set to the individual weights.

3.2 Physical Derivation

In the previous section the final LBM method was presented. However, to properly make use of the method, the physical significance needs to be understood. This section provides a first principles derivation of the method to show why it is a physically based method.

The desired output of our fluid simulation is a fluid surface. To move this we need to know the velocity of the fluid near the surface so that we can advect it for each time step. However, in order to achieve this, a velocity model of the entire fluid is required. The movement of a volume of fluid at the bottom of a container will cause a disturbance at the surface due to the attraction and repulsion forces between millions of nearby particles.

In section 2.4 two approaches to modelling the velocity, Eularian and Lagrangian, were described. The Eularian approach directly computes the macroscopic quantities for a give point in space, while the Lagrangian approach places particles in space with fluid like inter-particle forces. In physics the eularian approach came first in the form of the Navier-Stokes equations by Louis M. H. Navier (1785-1836) and Sir George Babriel Stokes (1819-1903) followed by the Lagrangian approach in the form of the Boltzmann equation [Wolf-Gladrow, 2000].

The LBM was created after many iterations of different lattice gas methods. It is possible to trace through each of these steps to obtain the current algorithm, but a greater understanding of how the algorithm works can be gained by its derivation from the physical equations. Here, the latter approach is taken, beginning with a grounding in some of the earlier work in physics.

The Boltzmann equation models the collisions of particles in a simplistic fashion, using three rules: (1) only two

University of Cape Town

particles are present in a collision, (2) the velocities of the particles are linearly independent and (3) there are no external forces present during the collision. The equation does not consider individual particles, but rather a distribution of particles, $f(\overrightarrow{x}, \overrightarrow{v}, t)$, which is defined such that $f(\overrightarrow{x}, \overrightarrow{v}, t)d\overrightarrow{v}d\overrightarrow{x}$ is the number of particles in a small portion of space, $d\overrightarrow{x}$, travelling in a small range of velocities, $d\overrightarrow{v}$ [Bhatnagar et al., 1954]. The distribution f is also known as the probability distribution of the particles and the integral of f over the whole space will yield the total number of particles. The time evolution of f for a given point, $\overrightarrow{x}$, and velocity, $\overrightarrow{v}$, is then given by

$$\frac{\partial f}{\partial t} + \overrightarrow{v}\frac{\partial f}{\partial \overrightarrow{x}} + \overrightarrow{g}\frac{\partial f}{\partial \overrightarrow{v}} = \Omega(f), \tag{3.11}$$

where $\overrightarrow{g}$ is gravity and Ω is the collision operator. In short, the equation captures the change in the particle distribution due to the distribution's spatial gradient at the position and velocity (the second term from the left), the acceleration of gravity (the third term) and lastly the inter-collisions of particles (the right hand side). The collision operator models all possible collisions between every pair of particles and is defined as

$$\Omega(f) = \int (f_1' f_2' - f_1 f_2)\sigma(|\overrightarrow{u_1} - \overrightarrow{u_2}|, \omega)d\overrightarrow{\omega}d\overrightarrow{p}.$$

This integrates the change in the distribution function due to all particle collisions, which always involve only two particles by assumption, and is performed by integrating over $\overrightarrow{p}$, the possible momentums of a single particle, and ω, the solid angle. This angle defines the velocity of the second particle, $\overrightarrow{u_2}$, relative to the first, $\overrightarrow{u_1} = \overrightarrow{p}/m$ (m is the mass of a particle). For a given pair of velocities, $\overrightarrow{u_1}$ and $\overrightarrow{u_2}$, the probability of a collision occurring is given by the differential cross section operator, σ modelled by approximating the particles as spheres [Thürey, 2003]. Now, once the probability of a collision is known, the change in the global distribution function f is the difference in the distribution functions before the collision, $f_1 f_2$, and after the collision, $f_1' f_2'$, calculated from the known particle velocities. The operator is included as it shows the complexity needed to model a particle system perfectly. For a more detail description the reader is referred to Succi [2001].

The main problem with the collision operator as it stands, is that it is very complex and computationally infeasible to model. Bhatnagar et al. [1954] introduced a new collision operator by relaxing the current distribution to an equilibrium distribution, f^e. This is performed using a collision time τ and the following equation,

$$\Omega(f) = \frac{f^e - f}{\tau}.$$

The approximation uses the H-theorem, which shows that f will tend to a Maxwellian distribution, f^e, to reach the lowest potential energy for the system [Thürey, 2003]. The collision time, τ, controls how fast the distribution will relax to equilibrium and f^e can be derived at the point where collisions have no effect on the distribution, i.e. $\Omega(f^e, f^e) = 0$.

Up to this point, the equations that model the underlying micro-dynamics of a particle system have been provided, but to measure the equilibrium of the system the macroscopic velocity, v_m, and density, ρ, are required. These, respectively, are an aggregation of the particle behaviour given by f as

$$\rho(\overrightarrow{x}) = \int f d\overrightarrow{v}, \quad \text{and} \tag{3.12}$$

University of Cape Town

$$\overrightarrow{v_m}(\overrightarrow{x}) = \frac{\int \overrightarrow{v} f d\overrightarrow{v}}{\rho(\overrightarrow{x})}. \tag{3.13}$$

The mass of the particle is normalized to unity in these equations. Now, with the macroscopic velocity and density, the equilibrium particle distribution at $\overrightarrow{x}$ is

$$g(\overrightarrow{v_m}) = f^e(\overrightarrow{v}, \overrightarrow{x}, \overrightarrow{v_m}) = \frac{\rho_o}{2\pi(RT)^{D/2}} e^{-\frac{(v - v_m(\overrightarrow{x}))^2}{2RT}}, \tag{3.14}$$

where ρ_o is the reference density (usually 1), R is the Ideal Gas Constant, D is the number of dimensions and T is the fluid temperature. This distributes the particles along velocities that agree most with the macroscopic velocity.

The Boltzmann equation gives us a link from the micro dynamics of particle interactions to the macro dynamics of fluid behaviour. The LBM uses this as a basis for fluid simulation. Using these tools, in the next section we will derive the LBM from the Boltzmann equation.

3.2.1 Lattice Boltzmann Method

The LBM is obtained by a special discretization of equation 3.11. Here, a derivation is presented. This is not as complete as texts such as Succi [2001] or Wolf-Gladrow [2000], but provides a correct derivation that is as simple as possible. The fluid space will be divided up into a grid, which allows the particle distributions to be tracked over time.

A finite set of velocities, $\overrightarrow{c_i}$, is introduced as the velocities along which particles will be allowed to travel. For each of these velocities, a corresponding discrete function, $f_i(\overrightarrow{x}, \overrightarrow{c_i}, t)$, is defined and Equation 3.11 with the simplified collision operator is now:[1]

$$\frac{\partial f_i}{\partial t} + \overrightarrow{c_i} \frac{\partial f_i}{\partial \overrightarrow{x}} = \frac{f_i^e - f_i}{\tau}. \tag{3.15}$$

The equation is further discretized with respect to time and space:

$$\begin{aligned}\frac{f_i(\overrightarrow{x}, t+\Delta t) - f_i(x,t)}{\Delta t} + c_{ix}\frac{f_i(\overrightarrow{x}+\Delta x, t+\Delta t) - f_i(\overrightarrow{x}, t+\Delta t)}{\Delta x} &+ \\ c_{iy}\frac{f_i(\overrightarrow{x}+\Delta y, t+\Delta t) - f_i(\overrightarrow{x}, t+\Delta t)}{\Delta y} &+ \\ c_{iz}\frac{f_i(\overrightarrow{x}+\Delta z, t+\Delta t) - f_i(\overrightarrow{x}, t+\Delta t)}{\Delta z} &= \frac{f_i^e - f_i(\overrightarrow{x}, t)}{\tau}\end{aligned} \tag{3.16}$$

As Δt and Δx in the discretization are arbitrary, $\Delta x = \Delta y = \Delta z = \Delta t$ is chosen. Substituting the 3 spatial differential terms on the left with an approximation of the discretized components into one term[2], leads to

$$\frac{f_i(\overrightarrow{x}, t+\Delta t) - f_i(x,t)}{\Delta t} + \frac{f_i(\overrightarrow{x}+\overrightarrow{c_i}\Delta t, t+\Delta t) - f_i(\overrightarrow{x}, t+\Delta t)}{\Delta t} = \frac{f_i^e - f_i(\overrightarrow{x}, t)}{\tau}. \tag{3.17}$$

[1]the gravity term has been excluded for simplicity and is not required, see section 3.1.2

[2]Instead of sampling f with individual steps along each of the axes and weighting the differences with the $c_{i\alpha}$, f is sampled at the full step along $\overrightarrow{c_i}$.

University of Cape Town

The last substitution we make is $\hat{\tau} = \frac{\tau}{\Delta t}$ and we multiply by Δt to get what is called the Lattice Boltzmann Equation (LBE):

$$f_i(\overrightarrow{x} + c_i\Delta t, t + \Delta t) - f_i(\overrightarrow{x}, t) = \frac{f_i^e - f_i(\overrightarrow{x}, t)}{\hat{\tau}}. \tag{3.18}$$

However, the equilibrium function, f_i^e, has not yet been defined. First, equation 3.14 needs to be transformed into a different form, which is obtained by the third order Taylor expansion of $g(\overrightarrow{v_m})$. Let $c_s = \sqrt{RT}$ and $\beta = \frac{\rho}{(2\pi c_s^2)^{D/2}} e^{-\frac{(\overrightarrow{v})^2}{2c_s^2}}$ then $g(\overrightarrow{v_m}) = \beta e^{-\frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2}}$. The important derivatives are then $\nabla g(\overrightarrow{v_m}) = \beta e^{-\frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2}}[-\frac{\overrightarrow{v_m}}{c_s^2} + \frac{\overrightarrow{v}}{c_s^2}]$, $g_{xx}(\overrightarrow{v_m}) = \frac{\beta}{c_s^2} e^{-\frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2}}[-1 + \frac{(v_{mx}+v_x)^2}{c_s^2}]$, $g_{yy}(\overrightarrow{v_m}) = \frac{\beta}{c_s^2} e^{-\frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2}}[-1 + \frac{(v_{my}+v_y)^2}{c_s^2}]$ and $g_{xy}(\overrightarrow{v_m}) = \frac{\beta}{c_s^2} e^{-\frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2}}[\frac{(v_x - v_{mx})(v_y - v_{my})}{c_s^2}]$. By the chain rule and differentiability of the exponential function, g is differentiable everywhere and the Taylor expansion around $\overrightarrow{0}$[3] up to second order is

$$\begin{aligned} g(\overrightarrow{v_m}) &= g(0) + \overrightarrow{v_m} \cdot \nabla g(\overrightarrow{v_m}) + \frac{1}{2}(v_{mx}^2 g_{xx}(\overrightarrow{0}) + v_{mx}v_{my}g_{xy}(\overrightarrow{0}) + v_{my}^2 g_{yy}(\overrightarrow{0})) + O(\partial^3 g(\overrightarrow{v_m})) \\ &= \beta[1 + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2} + \frac{1}{2c_s^2}(\frac{v_{mx}^2 v_x^2}{c_s^2} - v_{mx}^2 + 2v_{mx}v_{my}v_x v_y + \frac{v_{my}^2 v_y^2}{c_s^2} - v_{my}^2)] \\ &= \frac{\rho}{(2\pi c_s^2)^{D/2}} e^{-\frac{(\overrightarrow{v})^2}{2c_s^2}}[1 + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2} - \frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{(\overrightarrow{v_m}\overrightarrow{v})^2}{2c_s^4}] \end{aligned}$$

As each of our velocities will be discrete and the leading exponential will be constant, the simplified Maxwellian Distribution is

$$f_i^e(\overrightarrow{v_m}) = \rho w_i[1 + \frac{\overrightarrow{v_m}\overrightarrow{v}}{c_s^2} - \frac{(\overrightarrow{v_m})^2}{2c_s^2} + \frac{(\overrightarrow{v_m}\overrightarrow{v})^2}{2c_s^4}]. \tag{3.19}$$

The weight, w_i, is included to ensure that the new discrete distribution agrees with the continuous one for zero velocities. The simulations considered in this thesis are isothermal and, hence, c_s becomes an arbitrary constant [He and Luo, 1997].

When $\overrightarrow{v}$ or $\overrightarrow{v_m}$ are zero we have a new discrete distribution given by w_i. To ensure the consistency of this distribution, the concept of moments is introduced. The moments of a distribution function, f, around zero are defined as $\int_{-\infty}^{\infty} x^n f(x)dx$, where n denotes the nth-moment. For $n = 1$ the moment is just 1, for $n = 2$ the moment is the expected value of the distribution, for $n = 3$ the moment is a measure of the skewness of f and for forth moment it measures whether f is tall, skinny or squat. The moments of equation 3.14 are calculated using coordinate-wise Gaussian integrals while the discrete version given by w_i is a straightforward summation. The moments up to the fourth order are equated as below (from [Wolf-Gladrow, 2000]),

$$\sum_i w_i = \int g(\overrightarrow{v_m})d\overrightarrow{v_m} = 1 \tag{3.20}$$

$$\sum_i w_i c_{i\alpha} = \int v_{m\alpha} g(\overrightarrow{v_m})d\overrightarrow{v_m} = 0 \tag{3.21}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} = \int v_{m\alpha} v_{m\beta} g(\overrightarrow{v_m})d\overrightarrow{v_m} = p_o c_s^2 \delta_{\alpha\beta} \tag{3.22}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} c_{i\gamma} = \int v_{m\alpha} v_{m\beta} v_{m\gamma} g(\overrightarrow{v_m})d\overrightarrow{v_m} = 0 \tag{3.23}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} c_{i\gamma} c_{i\theta} = \int v_{m\alpha} v_{m\beta} v_{m\gamma} v_{m\theta} g(\overrightarrow{v_m})d\overrightarrow{v_m} = p_0 c_s^4(\delta_{\alpha\beta}\delta_{\gamma\theta} + \delta_{\alpha\gamma}\delta_{\beta\theta} + \delta_{\alpha\beta}\delta_{\beta\gamma}) \tag{3.24}$$

[3] $\overrightarrow{0} = (0, 0)$

University of Cape Town

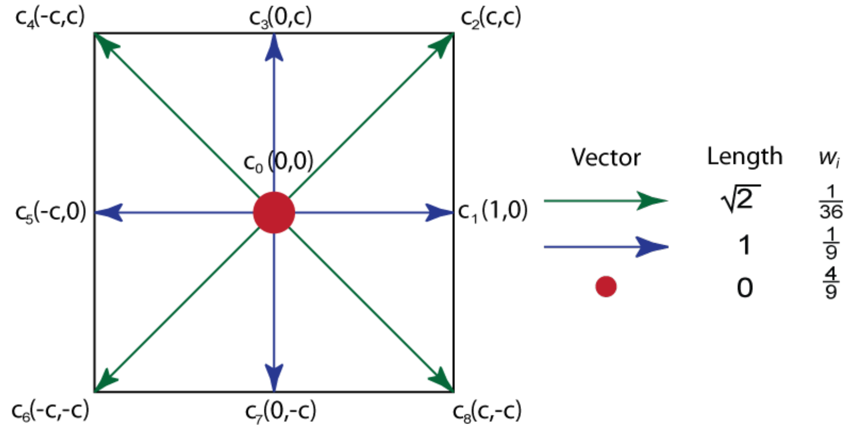


Figure 3.3: The 2D lattice.
There are nine velocities in total with the center dot giving the zero velocity. The velocities are labeled from the right horizontal position counterclockwise. Here, a constant c is used for the velocity components.

Where α, β, γ and θ take on possible co-ordinate values. Dirac's delta function is $\delta_{\alpha\beta} = 1 \Longleftrightarrow \alpha = \beta$ and zero otherwise. Now, given a set of finite lattice velocities, $\{\vec{c_1}, \ldots, \vec{c_n}\}$ we can use equations 3.18 and 3.19 to update a grid locations for each time step. Each of the grid locations stores f_i, the distribution corresponding to velocity c_i. For 2D, the lattice velocities are given in figure 3.3. We now calculate the zero velocity distribution, $\{w_1, \ldots, w_n\}$. Equations 3.21 and 3.23 have vanishing moments, so are not usable. Due to symmetry, each of the velocities with equal magnitude have equal weights, as such let $W_0 = w_0$, $W_1 = w_1 = w_3 = w_5 = w_7$ and $W_2 = w_2 = w_4 = w_6 = w_8$. From 3.20 we get

$$\sum_i w_i = W_0 + 4W_1 + 4W_2 = p_0 \tag{3.25}$$

From 3.22, for $\alpha = \beta = 1$ and $\alpha = \beta = 2$ we have

$$\sum_i w_i (c_{i1})^2 = 2c^2 W_1 + 4c^2 W_2 = p_0 c_s^2 \tag{3.26}$$

$$\sum_i w_i (c_{i2})^2 = 2c^2 W_1 + 4c^2 W_2 = p_0 c_s^2 \tag{3.27}$$

Note that terms where $\alpha = 1$ and $\beta = 2$ are not present as they cancel each other out due to symmetry. Lastly 3.24 gives us

$$\sum_i w_i (c_{i1})^4 = 2c^4 W_1 + 4c^4 W_2 = 3p_0 c_s^4, \tag{3.28}$$

and

$$\sum_i w_i (c_{i1} c_{i2})^2 = 4c^4 W_2 = p_0 c_s^4. \tag{3.29}$$

Solving these equations for W_i and c_s we get $W_0 = \frac{4}{9}$, $W_1 = \frac{1}{9}$,$W_2 = \frac{1}{36}$. As such, let $\Delta x = c_s \Delta t$. The constant, c_s, is known as the speed of sound [Körner et al., 2006], which is a measure of how quickly information (particles) can pass from one grid cell to the next. In general, Δt is normalised such that $c = 1$, which reduces the divisions in the equilibrium function, and then $\Delta x = \frac{1}{\sqrt{3}}$ [Thurey, 2007].

University of Cape Town

For the 2D lattice the velocity and density for a given grid point are calculated as:

$$\sum_{i=0}^{8} f_i = \rho \tag{3.30}$$

$$\sum_{i=0}^{8} \overrightarrow{c_i} f_i = \rho \overrightarrow{u} \tag{3.31}$$

The LBM for two dimensions is defined by the choice of $\overrightarrow{c_i}$ and the derived w_i. These are chosen with sufficient symmetry to allow the recovery of Navier-Stokes.

3.2.2 Proof of physical correctness

The full proof of correctness is too lengthy and detailed to present here. Instead, it has been included in Appendix B. In essence, the proof rests on the assumption that fluid and particle behaviour can be analysed on different scales. The distribution functions are decomposed into such different scales. The Taylor expansion of the LBE equation and the translations from the distribution functions to the macroscopic values of pressure and velocity in Equation 3.3 are then used to recover the Navier-Stokes equations. The most important result from the proof is the relation between the relaxation parameter, τ, and the viscosity of the fluid given by

$$v = \frac{c_s^2 \Delta t}{3} \left(\tau - \frac{1}{2} \right). \tag{3.32}$$

In the equation, $c_s = \frac{\Delta x}{\Delta t}$. In a normalised lattice with $\Delta t = \Delta x = 1$, this is

$$v = \frac{1}{3}(\tau - \frac{1}{2}). \tag{3.33}$$

3.2.3 Numerical Stability

In general, there has been little concrete research into on the stability of the LBM. The most thorough description of the topic can be found in Succi [2001]. The LBM is an explicit time-marching numerical scheme for the solution of the Navier-Stokes equations. Time-marching is used to describe a method that takes finite steps in time to solve a partial differential equation. Explicit refers to the information required to take a step forward in time. Specifically, only information from the previous time step is required.

A common condition when using such a scheme is known as the Courant-Friedrichs-Lewy (CFL) condition [Körner et al., 2006]. The basic premise is that the information at a point in a discretized structure cannot travel further then the neighbouring points in one time step. A simple example of such an inconsistency is a particle with velocity $2ms^{-1}$ traveling through a discrete grid with spacing of $1ms^{-1}$ and using a time step of $1sec$. Now, the particle will move two grid spaces in one step and will not be able to take into account information, say a collision, in the space between its start and end position. In the case of the LBM this condition is

$$v_{max} < \frac{\Delta x}{\Delta t} = c_s. \tag{3.34}$$

This says that the maximum velocity, v_{max}, must be less then c_s, the speed of sound of the lattice. After normalisation, $\Delta t = 1, \Delta x = \frac{1}{\sqrt{3}}$, the expression becomes $|v_{max}| < \frac{1}{\sqrt{3}}$. A similar relation suggested by Thürey

University of Cape Town

[2003] is obtained for gravity,

$$\frac{g(\Delta t)^2}{\Delta x} < 10^{-4}. \tag{3.35}$$

Here, the acceleration due to gravity is controlled. For our simulations this will be the only external force and the time-step and space-step must be chosen to ensure that the velocity of the fluid is not accelerated faster then the maximum velocity for the lattice.

Further analysis of the stability can be performed by studying the conservation of various physical quantities. The first being the viscosity of the fluid. This should never be negative and, hence, the restriction $\tau > 0.5$ is derived from Equation 3.33. The other quantities that should not be negative are the distributions and since they are further restricted from above, $0 < f_i,\ f_i^e < 1$ is obtained. Considering this and the equilibrium function we can see that $|v_{max}| < \frac{2}{3}$. Succi [2001] use these same restrictions to derive the constraint $\frac{u^2}{2} + \tau u_x < \frac{1}{3}$. This, however, is for a simplified 1 dimensional lattice, but does show that high speeds and a high velocity gradient will cause instabilities. They go one step further to suggest the relation $0.5 < h(\overrightarrow{u}) < \tau$ where h is some function of the current velocity that limits the value of τ. It is not yet known what the value of $h(\overrightarrow{u})$ is. Essentially, τ can approach 0.5, but cannot get arbitrarily close. Thurey [2007] use a cut off value of 0.51, obtained from experimental procedures.

Muders [1995] find this gradient to be a cause of "spurious instabilities." To combat this he introduces a parameter that increases the viscosity artificially in regions with velocities above a given threshold. He claims that this allows him to simulate fluids with speeds close to the speed of sound. In addition, he also uses a modified distribution function that takes into account a fourth order expansion of the Maxwellian distribution (in section 3.2 only order two was used). This modification uses 18 distinct equilibrium weights instead of our 3.

Another approach to increase the stability of the LBM is the Multiple-relaxation time method [d'Humières et al., 2002]. This method no longer uses the simplified relaxation operator of Bhatnagar et al. [1954], instead they use multiple relaxation times. This relaxes all the distribution functions at the same time. The collision operator becomes a function of all distribution functions instead of just the one previous distribution function and the equilibrium value. This moves the LBM towards an implicit method, relying on neighbouring distribution function values, increasing the stability at the expense of a more complex method.

Hou et al. [1994] claim that numerical instability in the LBM is caused by unresolved fluid motion at scales finer then the chosen LBM grid. They introduce a damping term for the effects created by sub grid motion, stopping the motion from causing unwanted instabilities. The damping term uses Smagorinsky's sub-grid model, also used in Thurey [2007] and Wei et al. [2003], which uses the local stress tensor as a measure of the sub grid turbulence at a point. For the LBM this is given by

$$\prod{}_{\alpha,\beta} = \sum_{i=1}^{19} c_{i\alpha} c_{i\beta} (f_i - f_i^e). \tag{3.36}$$

Then the intensity of the local strain is calculated as

$$S = \frac{1}{6C^2}\left(\sqrt{v^2 + 18C^2\sqrt{\prod{}_{\alpha,\beta}\prod{}_{\alpha,\beta}}} - v\right). \tag{3.37}$$

University of Cape Town

Here, C is the Smagorinsky constant, usually given a value in the range $0.2 - 0.4$ and v is the viscosity. The relaxation parameter is then modified to

$$\tau_s = 3(v + C^2 S) + \frac{1}{2}. \tag{3.38}$$

In effect this equation also artificially increases the viscosity of the simulation for points on the grid where there is higher sub-grid turbulence, damping the effects of the turbulence.

To summarise, there are two main points of instability: (a) when the fluid moves faster then some v_{max} and (b) when the velocity gradient is too large for the LBM to handle. These have been controlled by Thurey [2007] using adaptive time steps and the Smagorinsky model. We adopt the second approach, but not the first. High fluid speeds can be adequately dealt with by using sufficiently small time steps during the simulation, without added complexity overhead.

3.3 Conclusion

This chapter has introduced the LBM and the fundamental link to the atomic interactions via the Boltzmann equation and the discrete distribution functions on a lattice. The interactions with obstacles were also defined using a slightly modified no-slip condition. The evolution of the distribution functions was defined by the streaming and collision operators. The collision operator makes use of the Maxwellian distribution to correctly distribute the particle probabilities within a lattice cell for each time step. Through the use of a relaxation parameter, τ, the viscosity of the fluid can be controlled.

Lastly, the stability conditions for the LBM were discussed. Specifically, it is important to keep the spatial step over time step ratio greater then the maximum velocity (see Equation 3.34) and to limit the time step against gravity. To aid simulation stability, the Smagorinsky sub grid model was introduced.

University of Cape Town

Chapter 4

Single CPU Implementation

We aim to generate 3D free-surface fluid simulations for movies, integrating the simulation into the popular 3D animation package Houdini[1]. This will give an animator the ability to define a 3D scene and place fluid within it. The fluid will be simulated using the underlying Lattice Boltzmann numerical method, presented in Chapter 3. We now explore the fluid surface representation and how the fluid is embedded into the scene, beginning with an algorithm overview in Section 4.1, followed by the mass tracking or Volume of Fluid (VOF) method in Section 4.2.

The VOF provides a voxel representation for the presence of fluid in the scene and can be rendered using three dimensional texture techniques, but, for fluids such as water, it is preferable to have a polygonal surface that can be input to a renderer to produce effects such as refraction and reflection. The marching cubes methods is used to reconstruct a triangle surface from the VOF grid and is discussed in Section 4.3. The integration of the simulation into Houdini is discussed in Section 4.4. Lastly, the results from the single CPU fluid simulation and conclusions are presented in Sections 4.5 and 4.6.

University of Cape Town

4.1 Algorithm Summary

An overview of the VOF LBM algorithm appears in Figure 4.1. The LBM algorithm uses a voxel grid structure to simulate fluid in 3D during stream and collide (steps (1) and (2) in the figure), giving us the movement of the fluid for each position in the scene. To further track the movement precisely, we make use of the VOF method to track which grid cells contain fluid.

Each grid location, in Figure 4.1, shows the fluid region divided into four states: fluid, interface, empty and obstacle cells. The fluid cells represent cells that are completely filled with fluid and conversely empty cells are completely without fluid. Interface cells form the boundary between filled and empty cells and are partially filled. A filled cell must be surrounded only by fluid and interface cells, as fluid moves through the grid by exchanging mass with non-empty cells. This mass exchange is implemented in parallel using the stream step for efficiency. The Process Queues step is added to manage the filling and emptying of cells (step (3) in the figure).

[1] www.sidefx.com

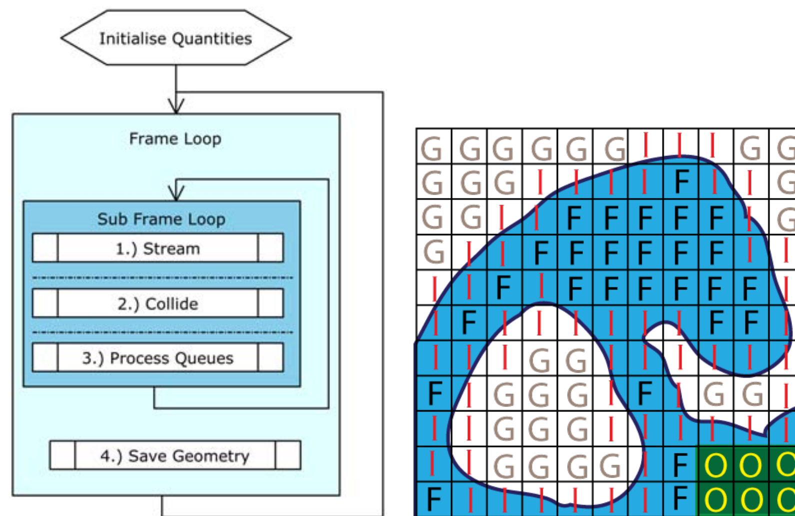


Figure 4.1: (left) The Volume of Fluid (VOF) LBM algorithm steps. (right) Scene cell division.
(left) The inner loop runs the simulation steps a number of times for each frame, as the time resolution required for stability is less than movie frame rates. The frame loop is run for as many frames are required for the final animation. Fluid physics and mass tracking are performed in the stream and collide steps, while Process Queues handles cell state changes.
(right) The fluid is divided up into fluid (F), interface (I), empty (G) and obstacle cells (O). All simulation of fluid is performed on either the fluid or interface cells, while the empty and obstacle cells are markers used for correct behaviour.

University of Cape Town

Finally, the fluid needs to be passed to the renderer in mesh format and this is implemented in the Save Geometry step (step 4). To simulate the underlying fluid dynamics, we run the sub frame loop a number of times for every frame saved, as the time step to maintain algorithm stability is less than the time step required for each frame rendered for an animation.

4.2 Volume of Fluid Method

This section defines the interactions between the different cell types and how the fluid moves through the scene. It is important to handle the boundary conditions and fluid gas interfaces correctly and the resulting problems and solutions are discussed. Specifically, the interactions of distribution functions to be streamed from empty cells must be reconstructed and the balancing of forces at the interface must be performed. There are also cases where interface cells can become isolated, so extra precautions must be implemented.

To keep track of the cell state a cell flag, a mass, $m(\overrightarrow{x})$, and a fill fraction value, $\epsilon(\overrightarrow{x}) = \frac{m(\overrightarrow{x})}{\rho(\overrightarrow{x})}$, are added to grid locations, with $\rho(\overrightarrow{x})$ being the density calculated from Equation 3.3. A fluid cell will always have a fill fraction of 1, while an empty cell has a value of 0. The interface cells occur in the range [0,1] and may change to the empty state, if $\epsilon(\overrightarrow{x}) < -0.01$, or to the fluid sate, if $\epsilon(\overrightarrow{x}) > 1.01$.

For each time step, the exchange of mass for a given cell with its neighbours is directly calculated from the LBM distribution functions, as they are a perfect description of the movement of fluid within a cell. The value of a distribution function gives the percentage movement of fluid along a given velocity. Analogously to the streaming operation, the distribution function, $f_i(\overrightarrow{x}, t)$, causes a proportional loss to the mass of the current

cell, while an adjacent inverse distribution function, $f_{\tilde{i}}(\overrightarrow{x}+\overrightarrow{c_i},t)$, causes a gain in mass. The change in mass along velocity i can then be computed as

$$\Delta m_i(\overrightarrow{x},t) = f_{\tilde{i}}(\overrightarrow{x}+\overrightarrow{c_i},t) - f_i(\overrightarrow{x},t). \tag{4.1}$$

For fluid cells the streaming operation applies unchanged, as the distribution functions already travel to neighbouring cells and fluid cells only share full boundaries with neighbouring fluid and interface cells. Interface cells on the other hand are partially filled and the mass exchange is proportional to the shared area with other interface cells, which can be approximated by the corresponding fill fraction average of the two cells [Thurey et al., 2005a]. The interface-interface cell mass exchange is given as

$$\Delta m_i(\overrightarrow{x},t) = \left(f_{\tilde{i}}(\overrightarrow{x}+\overrightarrow{c_i},t) - f_i(\overrightarrow{x},t)\right)\frac{\left(\epsilon(\overrightarrow{x}+\overrightarrow{c_i},t)+\epsilon(\overrightarrow{x},t)\right)}{2}. \tag{4.2}$$

It is only necessary to track the change of mass of fluid cells using Equation 4.1 and the interface-fluid cells using Equation 4.2. Combining all the changes for all velocities produces

$$m(\overrightarrow{x},t+1) = m(\overrightarrow{x},t) + \sum_{i=1}^{19}\Delta m_i(\overrightarrow{x},t). \tag{4.3}$$

In order to conserve mass, the mass exchange must be symmetric and the equations must be defined as such. This makes it necessary to store both fluid fraction and mass for each cell. The mass is updated during the stream step, while the fluid fractions are kept constant until after the mass exchange of all cells and then the updated density is obtained from the collision step.

Empty cells have no relevant physical quantities, besides air pressure. During streaming, interface cells that neighbour empty cells will not be able to copy the relevant distribution function, as it is undefined. In these situations, the distribution function is reconstructed and the streaming step (Equation 3.1) becomes

$$f_i(\overrightarrow{x}+\overrightarrow{c_i},t+1) = f_i^e(P_A,\overrightarrow{u}) + f_{\tilde{i}}^e(P_A,\overrightarrow{u}) - f_{\tilde{i}}(\overrightarrow{x}+\overrightarrow{c_i},t). \tag{4.4}$$

P_A is the pressure of the air; usually a reference air pressure of 1 is employed. Using Boyle's law we can translate the ideal gas pressure into density, which are the same in a normalised volume, $V=1$.

In cases where an interface cell is surrounded by a small number of empty cells, only a few of the distribution functions will be reconstructed, causing unequal forces on the surface boundaries. We follow Thurey et al. [2005a], who reconstruct all f_i that are in angular proximity to the surface normal. This is where the absolute value of the angle between the normal for the current cell, $\overrightarrow{n}(\overrightarrow{x}) = \frac{1}{2}\times\begin{pmatrix}\epsilon(x-1,y,z)-\epsilon(x+1,y,z)\\ \epsilon(x,y-1,z)-\epsilon(x,y+1,z)\\ \epsilon(x,y,z-1)-\epsilon(x,y,z+1)\end{pmatrix}$, and the discrete velocity, $\overrightarrow{c_i}$, is less then $\frac{\pi}{2}$. Otherwise stated,

$$\overrightarrow{n}(\overrightarrow{x})\cdot\overrightarrow{c_i} > 0. \tag{4.5}$$

The surface normal is approximated from the fluid fill fraction discrete gradient. After these reconstructions, the distribution functions will be smooth along the surface edges and all cells will have valid values for the streaming phases.

University of Cape Town

After streaming is complete, during the collide step, the fill fraction of interface cells are checked to see whether they have crossed the thresholds for emptying or filling and, if so, the cell is added to a filled or emptied queue. The state is not changed immediately, as this would prevent the fluid state from remaining constant during an entire update step. Instead, the Process Queues step is added as a post simulation step to handle state changes.

The Process Queues step needs to correctly remove and initialise new interface cells. To maintain a layer of interface cells surrounding the fluid cells, the filled queue is processed and surrounding empty cells are converted to interface cells. It is possible that one of the surrounding interface cells may be present in the emptied queue and must be removed, otherwise flickering of cell state will occur. Searching through the emptied queue for neighbouring cells is an expensive operation, so we introduce a queue position variable for each grid location. This increases the memory required for each cell by 2% (see Table 4.2 in Section 4.5). The position is initialised to -1 and when a cell is added to the emptied queue, its position in the queue is stored. Now, a check of whether a cell at a given position is in the emptied queue can now be performed in constant time, at the cost of more memory consumption. Next, the resulting emptied queue is processed and any fluid cell adjacent to a cell being emptied is converted to an interface cell.

Empty cells that are converted into interface cells need valid values for pressure, velocity and distribution functions. The average of neighbouring non-empty cell values are used to obtain estimates of the velocity and pressure, as cells will have locally similar physical characteristics. The distribution functions are then initialised using the equilibrium distribution at the average pressure and velocity. Fluid cells converted to interface cells retain their values, as the quantities are physically correct.

During conversion, the excess or negative excess mass, m_{ex}, above or below the thresholds is redistributed to neighbouring cells. For emptied cells, $m_{ex} = m(\overrightarrow{x}) < 0$, while for filled cells $m_{ex} = \rho - m(\overrightarrow{x})$. The redistribution strategy is important, as the direction of the distribution will be visible as fluid movement. The excess mass is shared amongst cells in the region of the interface normal [Thurey et al., 2005a], which is an approximation of the fluid movement direction, and weighted by the closeness of the normal and the direction towards the neighbouring cell, using

$$m(\overrightarrow{x} + \overrightarrow{c_i}) = m(\overrightarrow{x} + \overrightarrow{c_i}) + m_{ex} \cdot \frac{v_i}{v_{total}}, \tag{4.6}$$

where for filled cells,

$$v_i = \begin{cases} \overrightarrow{n} \cdot \overrightarrow{c_i}, & \text{if } \overrightarrow{n} \cdot \overrightarrow{c_i} > 0; \\ 0, & \text{otherwise}; \end{cases} \tag{4.7}$$

and for emptied cells,

$$v_i = \begin{cases} -\overrightarrow{n} \cdot \overrightarrow{c_i}, & \text{if } \overrightarrow{n} \cdot \overrightarrow{c_i} < 0; \\ 0, & \text{otherwise}; \end{cases} \tag{4.8}$$

while $v_{total} = \sum v_i$. The result of this operation is that the fluid surface is advected against to the normal for emptied cells and with the normal for filled cells.

A drawback to this VOF implementation is that the fluid only moves in and out of interface cells and the surface motion is equivalent to the creation and removal of these cells. Should an interface cell have no neighbours, it will be unable to move. Thus, there may be small static pieces of fluid left, isolated in the grid. For fluid movement to occur there must be two adjacent cells, so sheets or drops of water smaller than 2 grid cells cannot be simulated. To reduce these lone cells, Thurey et al. [2005a] try to enforce a single layer of interface

University of Cape Town

	standard cell at $\overrightarrow{x} + c_i$	no fluid neighbours at $\overrightarrow{x} + c_i$	no empty neighbours at $\overrightarrow{x} + c_i$
standard cell at $\overrightarrow{x}$	$f_{\tilde{i}}(\overrightarrow{x} + \overrightarrow{c_i}, t) - f_i(\overrightarrow{x}, t)$	$f_{\tilde{i}}(\overrightarrow{x} + \overrightarrow{c_i}, t)$	$-f_i(\overrightarrow{x}, t)$
no fluid neighbours at $\overrightarrow{x}$	$-f_i(\overrightarrow{x}, t)$	$f_{\tilde{i}}(\overrightarrow{x} + \overrightarrow{c_i}, t) - f_i(\overrightarrow{x}, t)$	$-f_i(\overrightarrow{x}, t)$
no empty neighbours at $\overrightarrow{x}$	$f_{\tilde{i}}(\overrightarrow{x} + \overrightarrow{c_i}, t)$	$f_{\tilde{i}}(\overrightarrow{x} + \overrightarrow{c_i}, t)$	$f_{\tilde{i}}(\overrightarrow{x} + \overrightarrow{c_i}, t) - f_i(\overrightarrow{x}, t)$

Table 4.1: Modified mass exchanges to reduce lone interface cells.
The VOF may leave some interface cells isolated, as they may not empty fast enough. To alleviate artifacts, the mass exchange for interface cells in 4.1 is modified for special cases. Interface cells that are either not adjacent to fluid cells or empty cells are forced to empty or fill faster by causing the appropriate cell to gain or loose mass. A standard cell is one which has both fluid and empty neighbours. Note again that the exchange is symmetric.

cells around fluid cells by modifying the mass exchange for interface cells without empty or fluid neighbours. Additionally, some lone cells are removed directly. Interface cells without empty neighbours are most likely to become fluid cells and are forced to fill faster, while those without fluid neighbours are forced to empty faster by substituting the normal mass exchange in Equation 4.1 with those given in Table 4.1.

It is an expensive operation to check whether a cell has empty or fluid neighbours, as 18 other cells have to be visited. We include three neighbour counts, one for fluid cells, one for empty cells and the last for interface cells. The counts are initialised at the start of the simulation and every time a cell state is changed, adjacent cells have their neighbour counts updated. This is yet another enhancement for speed at the expense of memory, but the counts are all `char` types and the memory footprint is small compared to the distribution functions, a less than 2% memory increase (see Table 4.2 in Section 4.5). The counts can then be used to determine which mass exchange to use for a given cell.

With the appropriate modifications and optimisations, the VOF algorithm described above provides a 3D scalar field of fill fractions. A scalar field assigns a real value to every point, $\overrightarrow{x}$, in space. In the following section, we use this scalar field to construct a triangle mesh that represents the surface of the fluid.

University of Cape Town

4.3 Fluid Surface Generation

The marching cubes algorithm was originally published by Lorensen and Cline [1987] and was created to extract a triangular surface from medical data consisting of a scalar field, F. They construct an iso-surface for various medical imaging techniques such as computer tomography (CT). An iso-surface is a surface where every point on the surface, S, has the same value in the scalar field, i.e. let $\alpha \in \mathbb{R}$ then $F(\overrightarrow{x}) = \alpha \, \forall \overrightarrow{x} \in S$. We make use of their algorithm to construct a fluid surface.

The scalar field from the VOF method provides an indication, via the cell fill fractions, of how much fluid each point of the scene contains. The surface of the fluid lies between the empty cells, with 0 fill fraction, and the fluid cells, with a fill fraction of 1. Hence, the fluid surface is appropriately represented by the iso-surface where the fill fraction is 0.5.

Marching cubes considers each cube with the corners of the cube corresponding to the center of the VOF LBM grid cells, as shown in Figure 4.2 (b). The state of the corners are marked as being inside the surface, for a fill fraction greater than 0.5, or outside the surface, for a fill fraction less than 0.5. The surface cuts an edge of the cube when the corner states are opposite. This defines $2^8 = 256$ possible ways that the surface

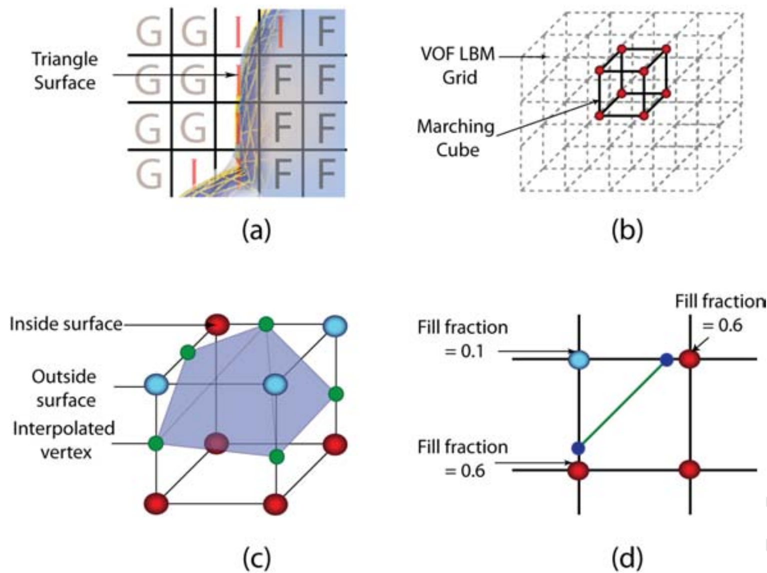


Figure 4.2: The marching cubes algorithm illustrated.
(a) The surface of the fluid will be triangulated on the fill fraction iso-surface of 0.5. (b) Each cube is considered in turn and classified by how the surface cuts the cube. (c) For a given cube classification, triangles are constructed and added to the final fluid mesh. (d) The position and normals of the triangle points are calculated as the linear interpolation of the positions and normals at the cube corners. The fluid surface will then be pulled to the position of the fill fraction of 0.5.

University of Cape Town

can pass through the cube. One example state is shown in Figure 4.2 (c), while the rest can be found in the original paper. For each case a shape is defined, from triangles, that will approximate the surface within the cube.

The vertices of the triangles lie on edges of the cube that are cut by the surface and the position of the vertices are calculated as a linear interpolation of the corner positions weighted by the fill fractions, so as to pull the triangle closer to the actual surface position. An example is shown in Figure 4.2 (d) for the 2D case. The normals at cube corners are approximated, as in Section 4.2, from the fill fraction discrete gradients, and the normal for a triangle vertex is the weighted linear interpolation of the normals at cube corners.

Marching cubes is implemented via a look-up table for the 256 different configurations with which the surface may cut a cube. For each cube, the corner states (inside or outside the isosurface) provide an index into the look-up table, which in turn provides the corresponding triangles that will be created. The interpolated values for the triangles are calculated and the triangles packed into an appropriate mesh.

We make use of the implementation provided by Lindh [2003], with modifications for our specific problem and some memory and speed optimisations mentioned in Lorensen and Cline [1987]. Naively, one can produce triangles for every cube and calculate the vertex position every time. However, each edge which the surface cuts will be shared with three adjacent cubes that will then share the same edge. We create an interpolated vertices (IVs) data structure, visualised in Figure 4.3, which stores two xz-planes of IVs. The IVs store the interpolated position and normals for edges. For each cube corner, we store three possible edge vertices along the x, y and z axes leading out from the corner. The marching cubes loop then starts by marking the interpolated vertices as un-calculated. When a vertex is needed, we see whether it exists in the data structure, if not, it is calcu-

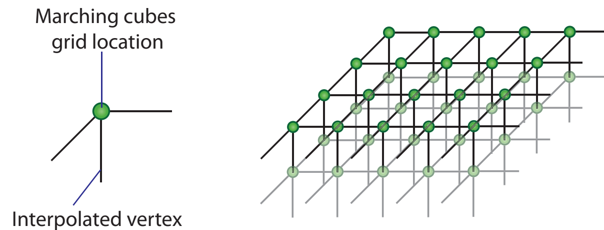


Figure 4.3: The Interpolated Vertices (IVs) structure used to optimise the marching cubes algorithm. On the left a single structure for a corner of the Marching cubes grid is shown. Each corner has three possible edges leading from it, that could be cut by the surface. We provide storage for each possibility. On the right, the two layers of IVs are shown. Only two layers are needed as the loops on other layers will not influence each other.

lated and stored, otherwise it is fetched. Additionally, the vertices or normals are created directly in the output mesh and the index is stored in the IVs. This creates a correct triangle mesh as output with no redundant items

Every second loop over the z index we re-use memory by re-initialising one plane of the IVs. Pseudocode for the algorithm is given in Figure 4.4.

The mesh created by triangulating the iso-surface has detail that is proportional to the resolution of the grid simulation. For grid-sizes of less then 40x40x40, we could produce fluid animations with surface reconstruction in real time, but since we are targeting film applications we will be using far higher grid resolutions. Thus, the mesh is saved to disk after reconstruction to be used at a later stage during rendering.

The marching cubes algorithm has a few shortcomings, summarised in Montani et al. [1994]. The three primary shortcomings are: topological ambiguities, computational efficiency and a large number of triangles created. The first arises in cases where a given cube, with corners marked as inside or outside the surface, has multiple possible surface configurations. This occurs when the underlying function is defined at a higher resolution than the grid used by the marching cubes algorithm. The result is that there will sometimes be holes in the constructed surface. A solution suggested by Ning and Bloomenthal [1993] is to sub-divide the grid into smaller cubes where ambiguities occur, until there are no longer any ambiguities. In our case, since we construct the surface on the same grid size as the scalar field, such ambiguities are less likely. The second problem (efficiency), is overcome by using data coherency of neighbouring cubes and parallelism. The target platform for our algorithm is a parallel environment, so this aspect will be taken into account, and data coherency is exploited in the IVs structure. The third problem is that a large number of triangles is produced for high resolution data sets. As the focus for this thesis is mainly on fluid simulation, rather than surface production, we will not apply any additional methods to reduce the number of triangles. But we note that there are optimisations to reduce the number of triangles that make use of adaptive techniques to reduce the sampling in areas of low detail or interest.

4.4 Houdini Integration

Thus far, a fluid simulation has been presented that is able to reproduce fluid behaviour, as given by the Navier Stokes equations (Chapter 3), and tracks the free fluid surface (Sections 4.1-4.3). To make more complex scenes, we need an interface that will allow specification of fluid and object positions, as well as modification

University of Cape Town

```
markIVsUncalculated();

for y = 1 to size_y

  for x = 1 to size_x
    for z = 1 to size_z
      set IVs[x][z][y+1%2] to uncaculated
    end
  end

  for x = 1 to size_x
    for z = 1 to size_z
      index = calculateLookUpIndex(x,y,z);

      for i = 1 to 12
        if surface cuts edge i
          if edge i not in IVs
            vertex[i] = interpolateValues();
            store vertex in IVs
          else
            vertex[i] = vertex from IVs
      end

      for each triangle in triangleLookupTable
        triangle = create triangle
        add triangle to mesh
      end
    end
  end

end
```

Figure 4.4: Pseudocode for the marching cubes algorithm used to generate the fluid surface from the VOF fill fractions.
The variable `triangleLookupTable` is a precomputed table with all possible ways a surface can cut a cube, which provides the triangles that must be created for the mesh at the cubes current position.

University of Cape Town

of parameters for the fluid simulation. An increasingly popular software package from SideFX software, called Houini, provides this interface.

Houdini aims to fulfill a unique role by producing dynamic and procedural effects, rather then frame by frame effects specified by an animator, effectively providing a higher level of automation and control. The power of computers to simulate individual objects and manage large amounts of detail is fully utilised to produce final 3D scenes that model human movement, rigid body interactions (modelling collisions and breaking objects) and procedural effects, such as L-systems [Lindenmayer, 1974], to name a few common applications. Artists set up initial conditions for a scene and Houdini creates the resulting animation for any number of frames. It fits into a production environment well, as it allows camera positioning and lighting specification. To render the scene, SideFX provide Mantra, a ray tracer packaged with Houdini. Animators are then able to produce high quality visual effects with physically realistic renderings.

Houdini is designed from the ground up to allow technical directors to produce visual effects for movies and advertisements. They aim to make the scene specification as dynamic as possible and have released the Houdini Development toolkit (HDK) to allow developers to create plug-ins for Houdini. The toolkit provides some basic types, such as geometric primitives, and operations, such as loading and saving geometry to file. In addition, very powerful operations are included, such a ray casting. Custom plug-ins allow control over creation of geometry or particles that interact in a manner that can be specified in C++ or Python.

The required structure and workflow for Houdini is quite different to other software with similar applications. In the following sub-sections, we will discuss this and which features of the HDK were used to specify and run

fluid simulations. The last section covers the pre-processing necessary for scene specification, before running the simulation.

4.4.1 Houdini Structure

Figure 4.5 illustrates the basic Houdini interface. This has a scene editor structure that is similar to other 3D packages such as 3D Studio Max[2]. Using the quick links tools and the scene view, one can create basic geometric primitives and manipulate objects in the scene into a desired arrangement. The scene can be rendered by specifying a camera and lights, which are passed to Mantra.

The power of Houdini lies in its scene specification using operator nodes (Figure 4.6 gives a close up view of the network pane). The operator nodes have data inputs and outputs and define operations on any type of data. A simple example is the union operator. The inputs here are two geometric types combined using the regularised union boolean operation [Foley, 1995] to produce one single piece of geometry. The flow of data from one node to another can be controlled via connection lines, making scene organisation straightforward and quick to change. Each operator corresponds to a C++ class that is compiled as a library, either statically or dynamically linked, that is then loaded by Houdini.

A custom operator is added to Houdini by creating a C++ class using the framework provided in the HDK. Within the C++ specification, parameters are defined for custom operators, which then appear on the parameter pane when the operator is loaded by Houdini. These parameter values can then be altered by the user easily, even on a frame by frame basis.

There are a number of abstract classes provided by the HDK for different applications. One example is the Surface Operator or SOP base class, which is used to define any operators that modify or create surface geometry. To create a new operator, the custom class is inherited from one of the base classes and the appropriate virtual functions are overloaded. When necessary, Houdini calls a virtual function defined for each of the base classes to update the output data based on the input data to the operator. Unfortunately, this function is not standardised across all the operator types. The custom class then makes calls to Houdini to fetch the current parameter values and any input objects that are relevant to the operators functionality.

University of Cape Town

4.4.2 The VOF LBM Custom Operator

Dynamic simulations are handled via dynamic operators, which define how objects in a scene are controlled by one or more solvers. The solvers use simulation methods to alter and control the position of the objects. We create a custom operator, a *VOF_LBM_Solver* class, that appears as an operator, as shown in Figure 4.6. To make scene objects influence the fluid simulation, features are passed to the solver via merge operators. The five features that we define are: (1) constraints, (2) pumps, (3) collision objects, (4) sources and (5) sinks. Note that additional cell states are defined for the pump and sink features.

The constraint feature is used to define the fluid domain and is given by a cuboid. The VOF LBM grid is then aligned with the position of the cuboid and the other features are defined relative to the constraint cuboid.

[2] `http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112`

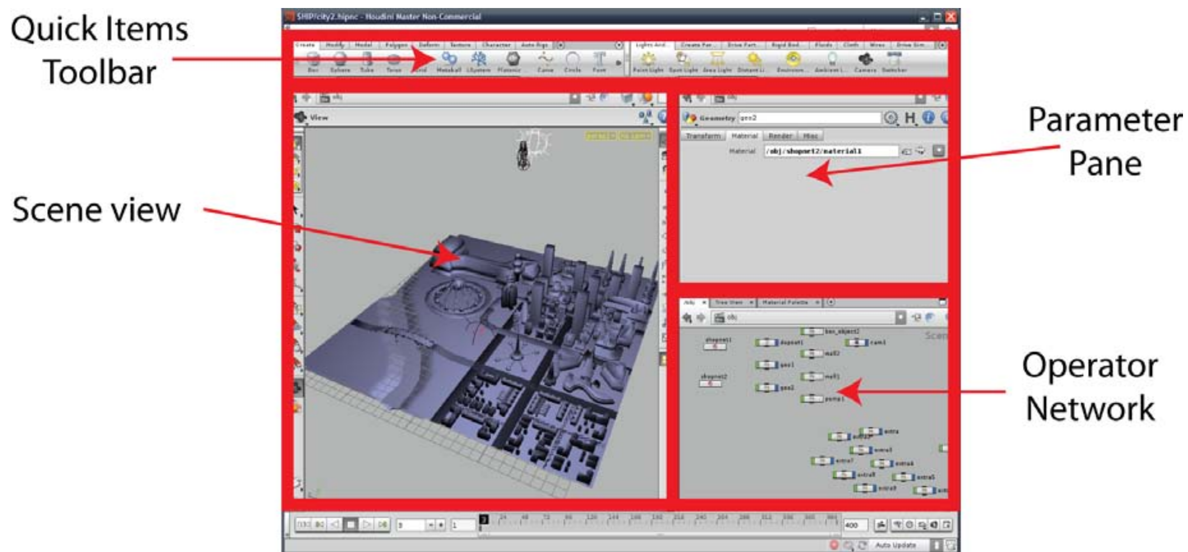


Figure 4.5: The basic Houdini interface.
The interface consists of the scene view, operator network and parameter panes, as well as the quick items toolbar. The view pane allows manipulation of the objects in the scene, the parameter pane the choice of parameters for the selected object and the operator network is where the data work flow can be linked together. Each of the operators can have many data inputs and outputs, producing new data or editing current data that is passed in. At the bottom of the screen, the time-line of the simulation is shown, allowing viewing of any frame of the animation and playback options.

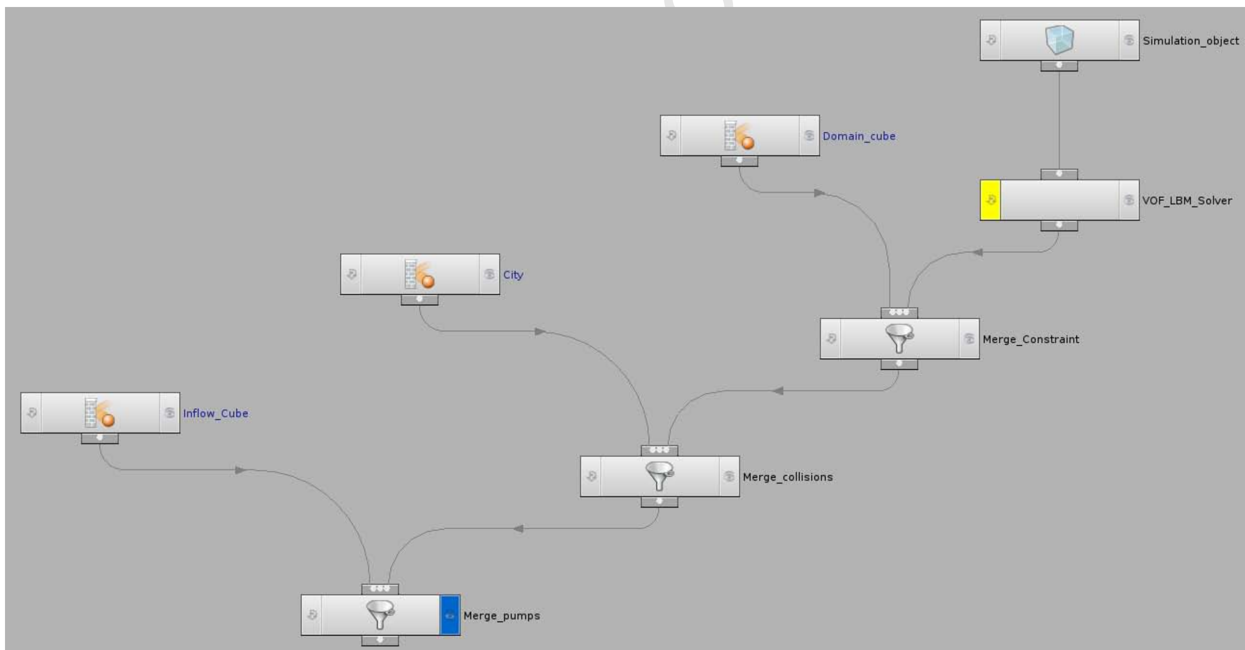


Figure 4.6: An example of the dynamic fluid network.
All simulation data is attached to the Houdini defined *Simulation_Object*. For our fluid simulation, the fluid geometry will be created for every frame and can be attached to this object. Due to large memory requirements, we save the geometry to disk instead of using this feature. The *Simulation_Object* is passed as an input into the *VOF_LBM_Solver* custom operator, activating the fluid simulation. The constraint, collisions and pump features are added to the simulation via the merge operators, which take the current simulation as input and the 3D object used as the feature. The inflow cube, city and domain cube can be part of a rigid body solver that causes collisions to occur with other rigid bodies, as well as the fluid.

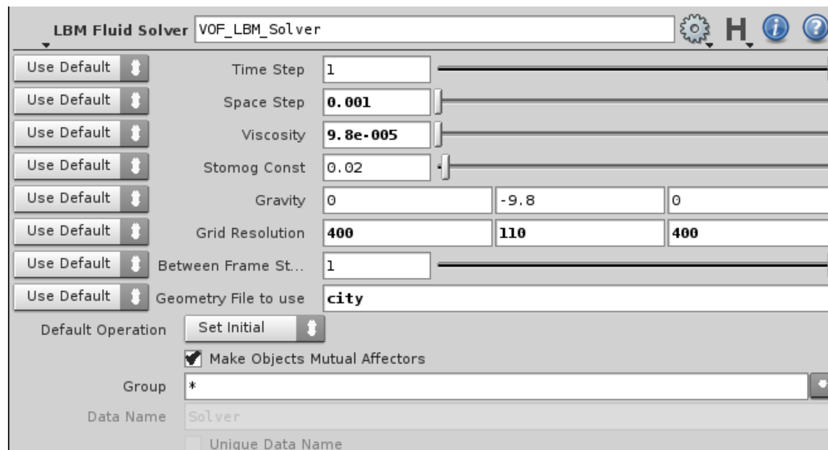


Figure 4.7: The parameters pane for the VOF LBM custom operator.
From the top: The *time step* is not the actual time step. Rather, the *time step* as calculated in Section 3.1.3 is divided by this value, to allow control over the time resolution. *Space step* gives the width of the grid cells and *Viscosity* is self explanatory. The *Stomog Const* is used in the sub grid turbulence model, while *Gravity* accelerates the fluid in the specified direction. *Grid Resolution* controls how detailed the simulation is. *Between frame steps* is the number of simulation interactions between steps of surface generation and *Geometry* file specifies where to save the current fluid surface.

University of Cape Town

The pump feature is used to allow fluid to flow into the scene from an outside source, such as water from a shower head, and the velocity of the object is used as the initial velocity for the fluid. The collision objects form the solid obstacles within the scene. The source objects define the initial still bodies of water, such as a pool of water, and the sink feature allows the absorption of water from a scene, to prevent a scene becoming totally filled.

To allow scene scrubbing, that is, the process of quick viewing of different frames of the scene, Houdini saves all simulation data created for every frame by making a copy of the solver class. In some instances, this would be very useful, as the simulation state would be available for every frame of animation. Modifications to the result of the simulation could then be made after a specific number of frames, but, due to the large size of the VOF LBM grid, this is not possible as animations typically consist of hundreds of frames, causing memory restrictions. Instead, we create a static fluid solver class to save memory, reducing flexibility, but making it possible to produce simulations with higher resolutions. In addition, the geometry format of Houdini consumes large amounts of space and the fluid geometry for each frame of animation is saved to disk using the HDK functionality. A simple load geometry operator is then used to view the fluid.

We also found that Houdini does not have a compact mesh representation, as it only saves individual triangles as opposed to vertex and edge lists, commonly used in more compact mesh representations. This made it necessary to store the fluid mesh for each frame in our own format, which can be converted to the binary geometry format needed by Houdini.

To control the fluid simulation, the *time step*, *space step*, *viscosity, Smagorinsky constant, gravity, grid resolution, iterations between frames* and *geometry file* parameters are defined, as shown in Figure 4.7.

4.4.3 Scene Initialisation

To run a simulation the parameters and features provided via the Houdini user interface must be translated into VOF LBM grid information. First, the domain constraint is read from the constraint rectangle. From this and the grid resolution, we calculate the step values used to find the absolute position of each grid location in the scene. Each grid location, at index (x, y, z) in the grid, is then compared against the feature objects to initialise the cell to the appropriate state by assigning obstacle objects, pump objects, sink objects and the source objects (in that order). A precedence of cell states is defined with obstacle cells taking the highest. The lowest priority is given to source cells. After the cell states are initialised, the appropriate physical quantities for each of the states need to be initialised in order to run the simulation.

The first important physical effect, not mentioned in the graphics literature, but common knowledge in physics, is that fluid pressure varies linearly with depth when the fluid is placed in a container. A drop of water in mid air will not have such a relation, as there is virtually no resistance from air. The pressure, P, at a depth h in the container is given by

$$P = \rho g h. \tag{4.9}$$

For each fluid cell, we check to see if there are no empty cells between it and an obstacle cell below, signaling that the fluid cell has a floor below and is held within a container. This assumes that there will always be sides to the container. If this were not the case, the relation would no longer be linear, but there would still be a pressure gradient. The pressure for a fluid cell with a floor is then set to the pressure of the fluid cell above it, plus the increment given by equation 4.9 adjusted to the lattice normalised values. If no fluid cell is present above a fluid cell or for fluid cells with no floor, the pressure is set to 1.

The pressure values of pump cells are set to slightly above 1, as they will distribute fluid into a scene and the higher pressure values force out more fluid. This parameter can be changed to achieve different simulation effects. Lastly, the pressure of the sink cells is set to 1.

The velocity of all cells is initialised to the velocity of the feature supplied to the VOF LBM operator, which in most cases is zero unless the feature is a pump, in which case the velocity is often specified. Once we know the velocity and pressure of each cell, we can use the equilibrium function to set the distribution functions to the appropriate values.

The mass of all cells is set to 1, unless they are interface cells, in which case a value of 0.4 is used, or sink cells, which have 0 mass. This value is chosen arbitrarily and any value may be chosen. During the simulation, the mass of sink cells is left constant at zero to absorb as much fluid as possible during mass exchange.

4.5 Animation Results

Having created a single CPU implementation that is integrated into the Houdini software package, we now provide an analysis of the effects that can be achieved using this software. The influence of different parameter values on the simulation is also explored. The test case considered is a drop falling into a pool. Other tests cases are examined in the final parallel implementation results in Chapter 6. However, this single test case is

University of Cape Town

sufficient to analyse the simulation effects.

In the following subsections, we consider the effect of varying the spatial resolution (Δx), the Smagorinsky constant *(S),* the time step (Δt), grid resolution (G) and viscosity (v). Grid artifacts are also considered and, lastly, some performance analysis. Note the spatial resolution is the distance between two cells (on one axis from center to center), and is different from the grid resolution, which is a measure of simulation scale.

Unless otherwise stated viscosity is set to 1×10^{-5} (the viscosity of water) and gravity to $9.8ms^{-1}$.

4.5.1 Spatial Resolution

For the falling drop, the spatial resolution is varied, while the other parameter values are held constant, as given in Figure 4.8. The entire animation is 72 frames in length and frame 20 is shown in the figure for different values of Δx.

The simulation is unstable and very bumpy surfaces are produced until the space resolution reaches $0.0006m$. After this point, turbulent effects are captured well, but ultimately the effects are smoothed out at very low scales. For subsequent tests a spatial resolution of $0.0004m$ is used as this gives sufficient turbulent behaviour, while avoiding surfaces that are too bumpy. The VOF method also produces bumpy artifacts, as the surface is moved within a grid and, thus, low grid resolutions will cause additional surface perturbations.

The reasons for this stability behaviour can be seen in the calculation of $\frac{1}{\tau}$, the relaxation parameter. Now, $\tau = 3v\frac{\Delta t}{(\Delta x)^2} + \frac{1}{2}$, with v being the viscosity (see section 3.2.3). Therefore, as Δx gets larger so τ approaches $\frac{1}{2}$, the region of instability for the LBM [Succi, 2001].

The VOF LBM method presented in this thesis has been shown to have to have a small region of stability with respect to spacial resolution for the viscosity of water. Thus, water needs to be simulated at a scale of 1mm-6mm. For different viscosities, the exact values are determined by the stability conditions. Most of the simulations for film applications require large amounts of fluid, so larger grid sizes will be required for such applications. Sometimes large scales may be required, however, for such applications, as long the effect looks good, it is usable. Water behaviour has many similarities at different scales, and one scale can be used to approximate another.

University of Cape Town

4.5.2 Smagorinsky constant

Four out of five parameters are kept constant, the values used are provided Figure 4.9, while the value of the Smagorinsky constant is varied. In the figure, frame 27 of the animation was used, as it illustrates the differences due to the constant more effectively than frame 20.

Higher values for this constant smooth out the fluid behaviour, as the viscosity is artificially increased in regions of high fluid stress, allowing the LBM to remain stable. Again, the appropriate value is not fixed and artists can create different effects by modifying this parameter. For further analysis, the constant value is set at 0.03, as recommended by Thurey [2007].

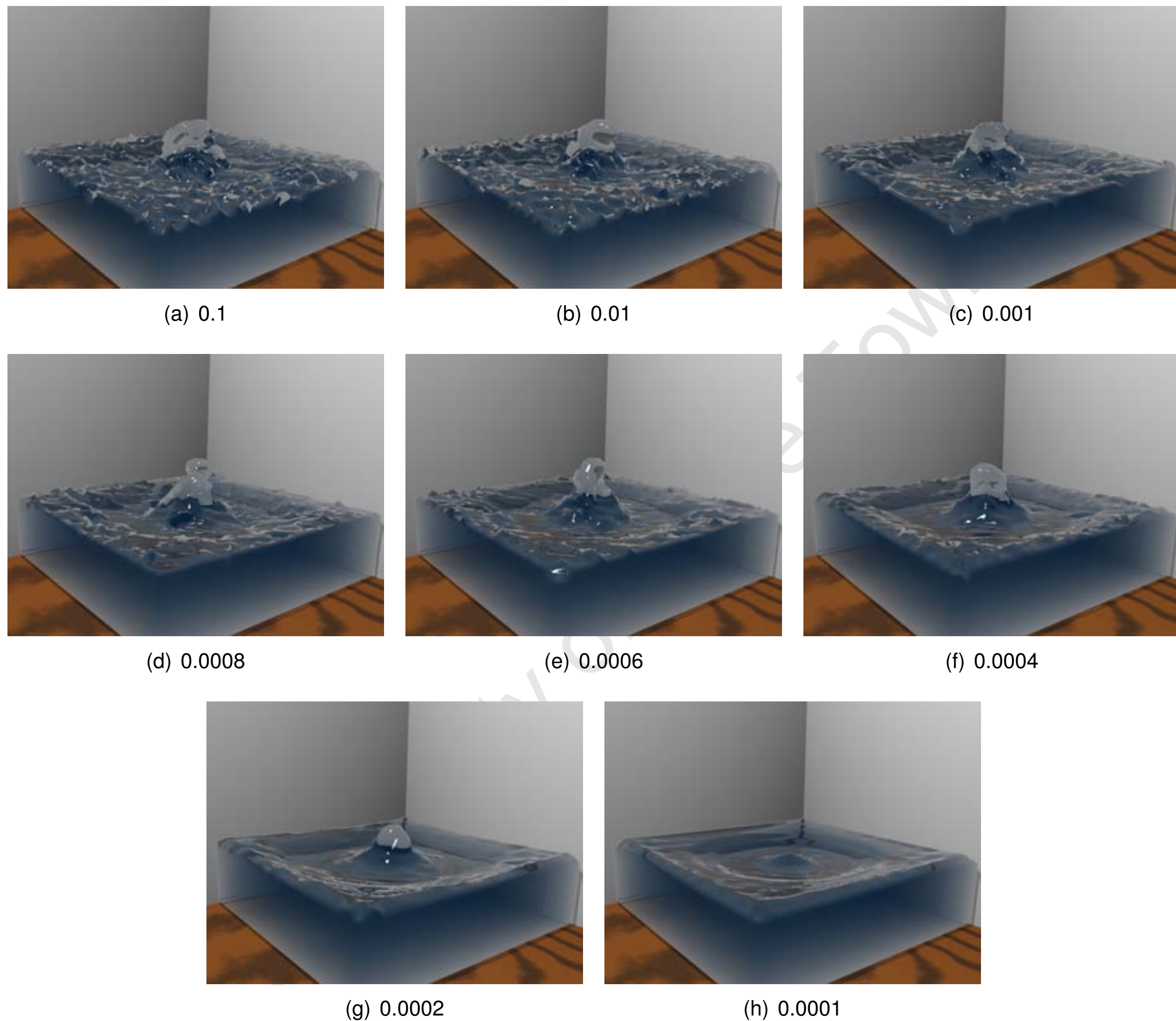
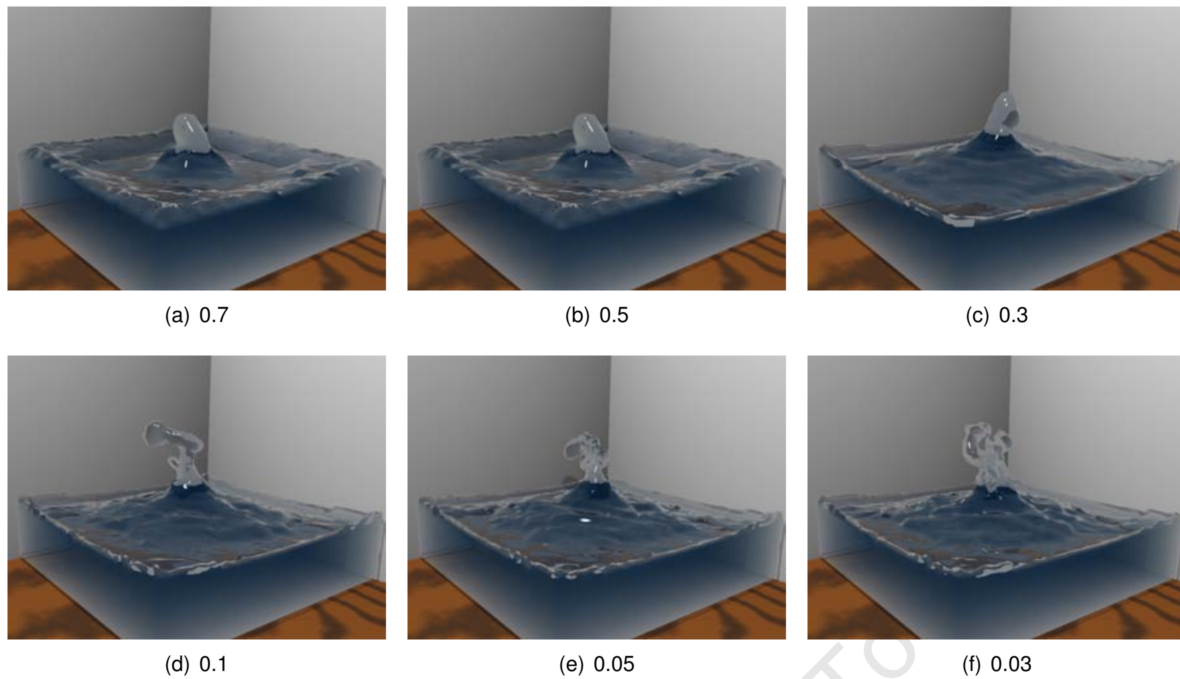


Figure 4.8: The falling drop at different spatial resolutions ($\Delta t = 0.05s$, $S = 0.03$ and $G = 50 \times 50 \times 50$). The spatial resolution is given below the appropriate sub-figure, while other parameters are constant. The simulation is unstable and very bumpy surfaces are produced until the space resolution reaches $0.0006m$. After this point, turbulent effects are captured well, but ultimately the effects are smoothed out at small spatial resolutions.



(a) 0.7 (b) 0.5 (c) 0.3

(d) 0.1 (e) 0.05 (f) 0.03

Figure 4.9: The falling drop for different Smagorinsky constants ($\Delta t = 0.01s$, $\Delta x = 0.0004$ and $G = 50 \times 50 \times 50$). All parameters are kept constant, while the Smagorinsky constant is shown for each sub-figure. Higher values damp the fluid turbulence and smooth the fluid surface.

University of Cape Town

The constant provides a useful method for controlling the stability of a simulation. For example, if the simulation is very unstable at a spatial resolution of $0.0001m$, then by increasing the Smagorinsky value, one can still recover stability. However, high values often produce unwanted results as the turbulence is no longer allowed to occur.

4.5.3 Time Step

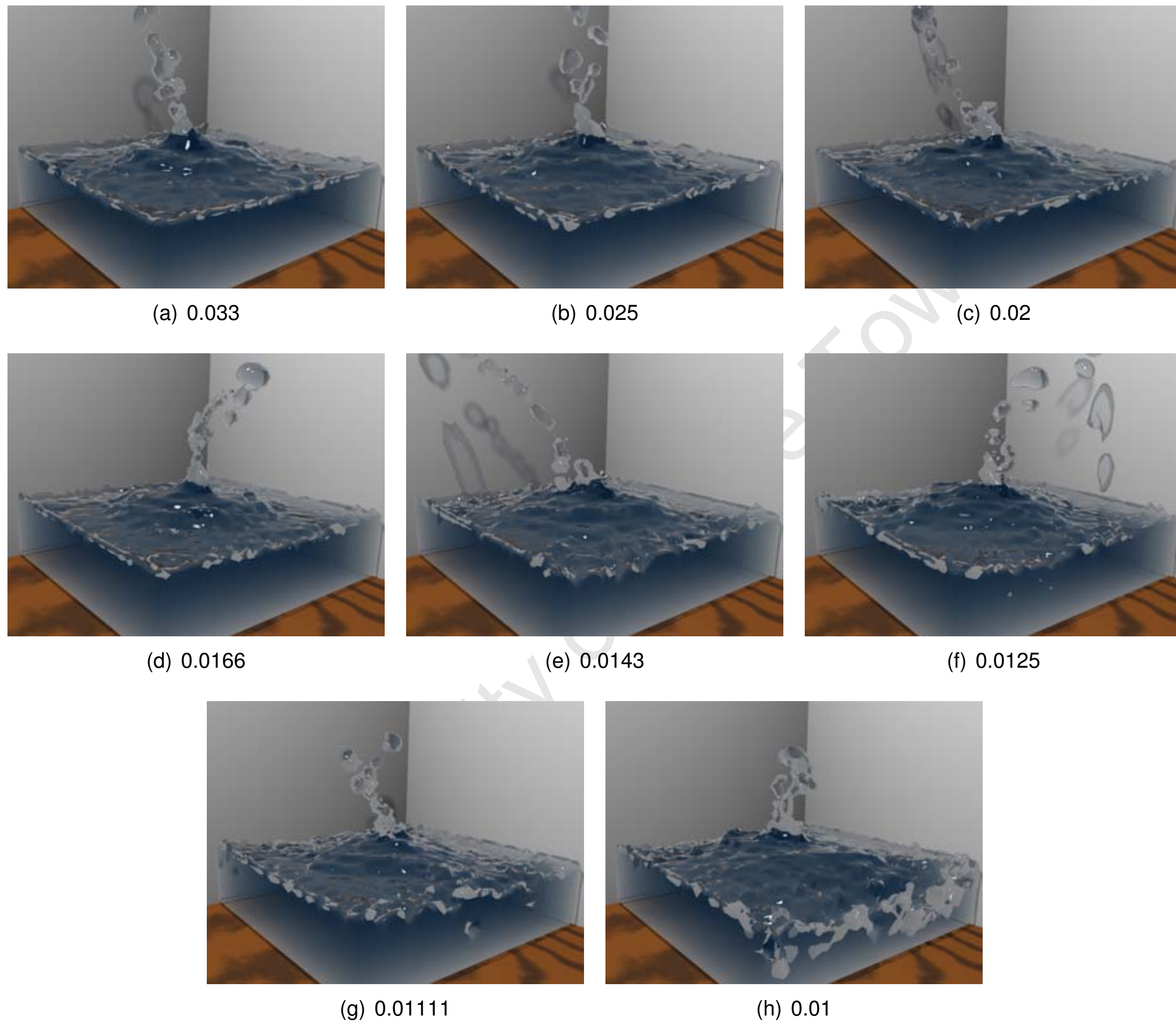
Again, four out of five parameters are kept constant, as shown in Figure 4.10, while the value of the time step is varied. Frame 27 is once again used as the example frame.

The figure shows that with smaller time-steps, more detail is captured, but that ultimately the simulation becomes unstable. Thurey [2007] claim that using the Smagorinsky sub-grid model to increase stability allows an arbitrary choice of time step, but this is not the case, as demonstrated.

The time step creates this instability in the same way as the space step in Section 4.5.1. As Δt tends to zero, so τ tends to $\frac{1}{2}$.

4.5.4 Grid Resolution

Two out of the five parameters are kept constant, as shown in Figure 4.11, while the simulation is run for three different resolutions: (1) $50 \times 50 \times 50$, (2) $100 \times 100 \times 100$ and (3) $200 \times 200 \times 200$. Every sixth frame of the animation is shown. As the domain is cubic, the resolution is increased in proportion to retain the symmetry,



(a) 0.033 (b) 0.025 (c) 0.02

(d) 0.0166 (e) 0.0143 (f) 0.0125

(g) 0.01111 (h) 0.01

Figure 4.10: The falling drop for different time steps ($\Delta x = 0.0004m$, $S = 0.03$ and $G = 50 \times 50 \times 50$). Four out of five parameters are kept constant, while the time step is shown for each sub-figure. A smaller time step per LBM iteration allows more detail to be simulated, but the fluid becomes more energetic and finally for very small time steps, it becomes unstable.

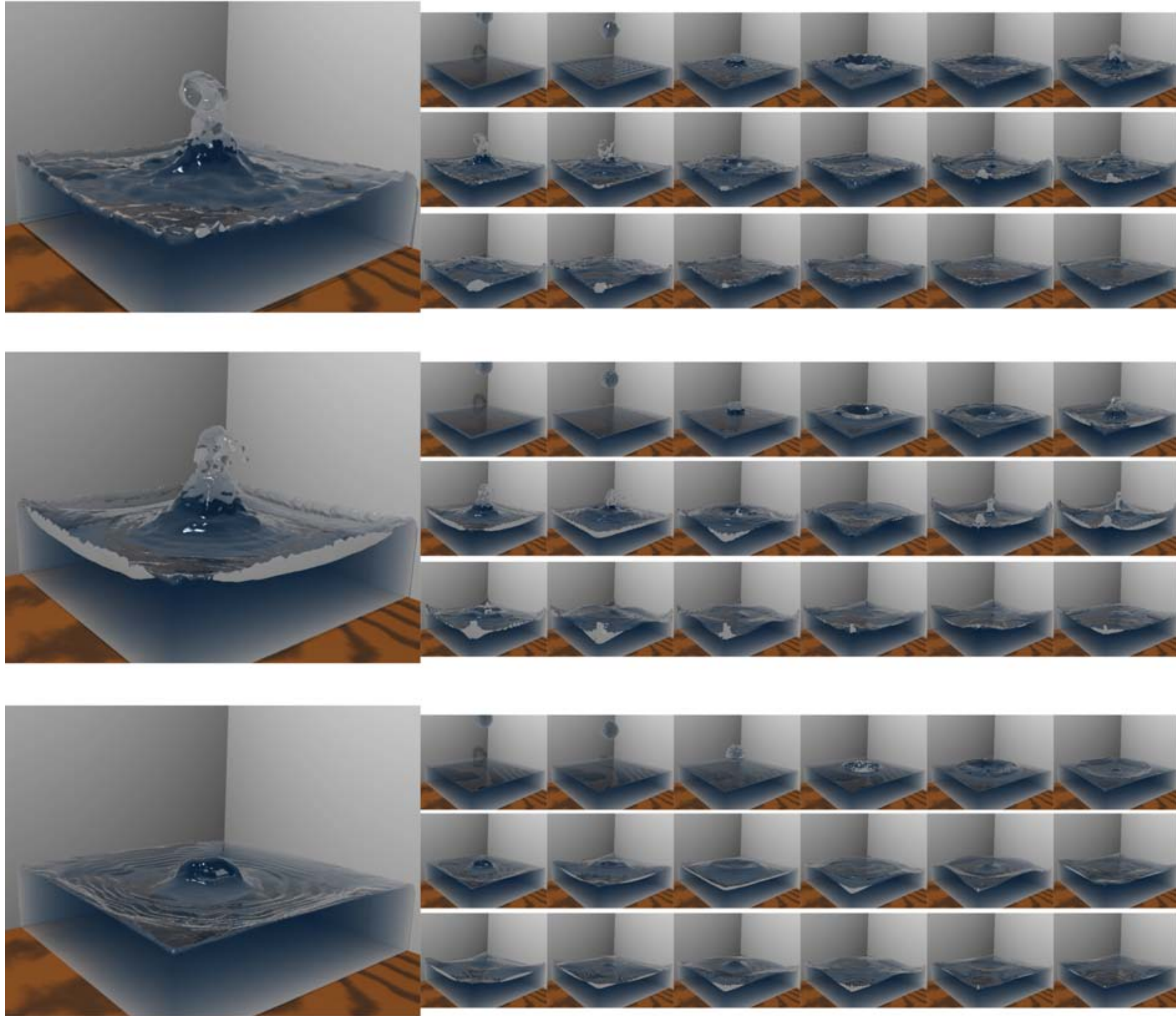


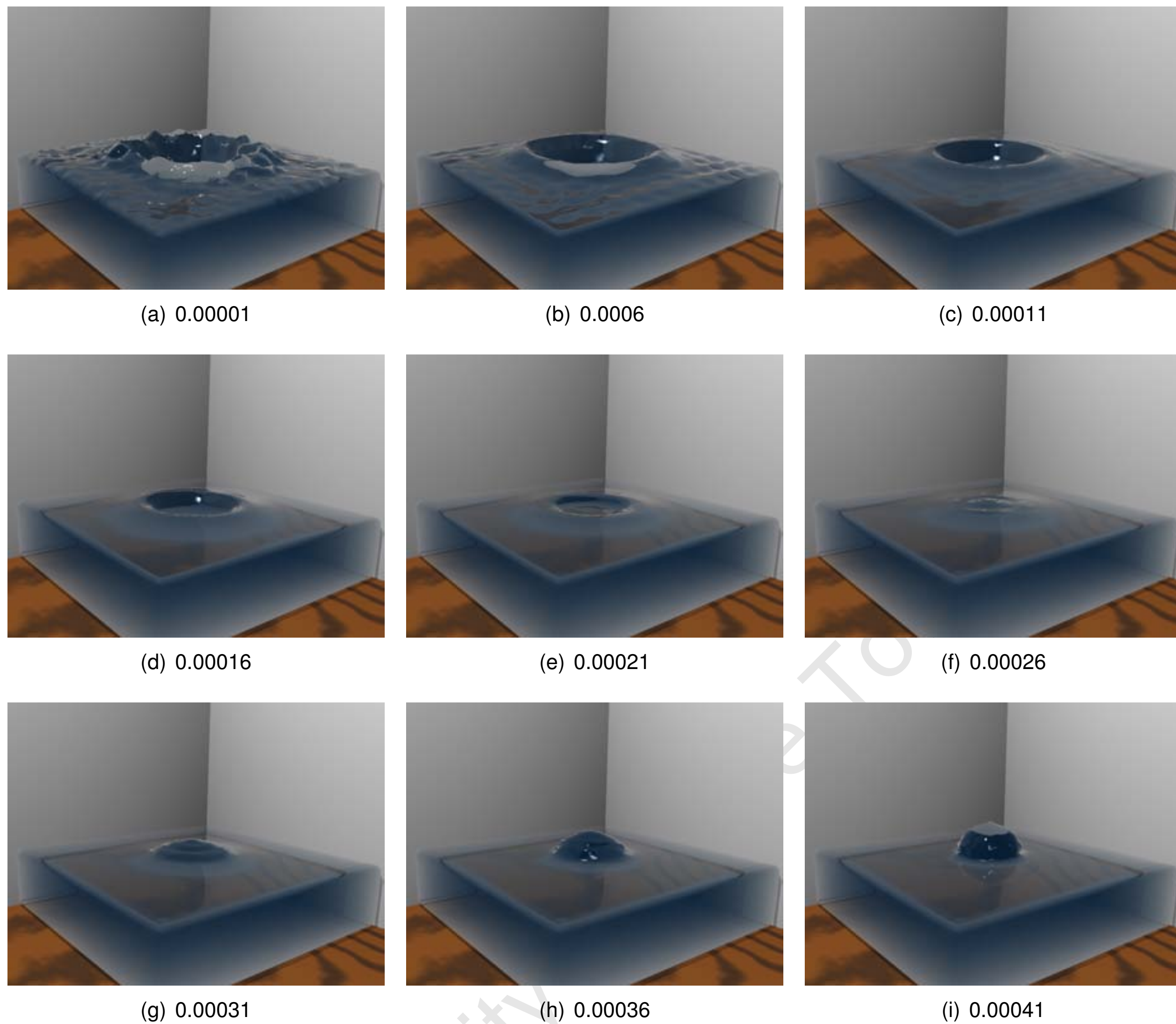
Figure 4.11: The falling drop for different grid resolutions ($v = 0.00001ms^{-1}$ and $S = 0.03$). The two parameters above are kept constant, while the top row of images is at a grid resolution of $50 \times 50 \times 50$, the middle at $100 \times 100 \times 100$ and the bottom row at $200 \times 200 \times 200$. The spatial resolution is chosen as $\Delta x = \frac{2cm}{k}$, where $k = 50, 100$ or 200 for the respective case and the time step is chosen to maintain stability.

since cuboid water drops do not look realistic.

The width of the domain is kept constant at $2cm$, so the space step for the three resolutions is $0.0004m$, $0.0002m$ and $0.0001m$, respectively. As shown in Section 3.1.3, the time step is dependent on the space step and, as such, the two higher resolutions had to be run for double the number of simulation iterations to keep the time step below the limit for stability. The higher resolution shows the fluid behaviour being smoothed away due to the small scale required, but higher surface detail is evident (in the complex surface ripples).

4.5.5 Viscosity

Four out of five parameters are kept constant, see Figure 4.12, while the value of the viscosity is varied. This time, Frame 13 is shown, as for high frame numbers only the container fluid is visible and none of the splashing



(a) 0.00001 (b) 0.0006 (c) 0.00011

(d) 0.00016 (e) 0.00021 (f) 0.00026

(g) 0.00031 (h) 0.00036 (i) 0.00041

Figure 4.12: The falling drop for different viscosities ($\Delta t = 0.05s$, $\Delta x = 0.0004m$, $S = 0.03$ and $G = 50 \times 50 \times 50$). Four out of five parameters are kept constant, while the viscosity is shown for each sub-figure. Larger viscosities smooth the fluid as expected.

effects. This is due to the smoothing effects of high viscosities.

Turbulence can be seen for low viscosities in Figure 4.12 (a)-(c) and the water behaviour becomes far calmer for viscosities of 0.00016 and up, (d) - (i). In the last sub-figure, (i), the fluid is very slowly absorbed into the pool and forms a small bump on the fluid surface.

4.5.6 Grid Artifacts

A scene is usually represented using a collection of geometric primitives and parametrically defined objects. These must be translated into the VOF LBM grid structure to be simulated correctly. Figure 4.13 shows that details cannot always be resolved. This could be fixed by using higher grid resolutions, or by making every triangle of a mesh in Houdini resolve to at least one grid cell in the LBM grid.

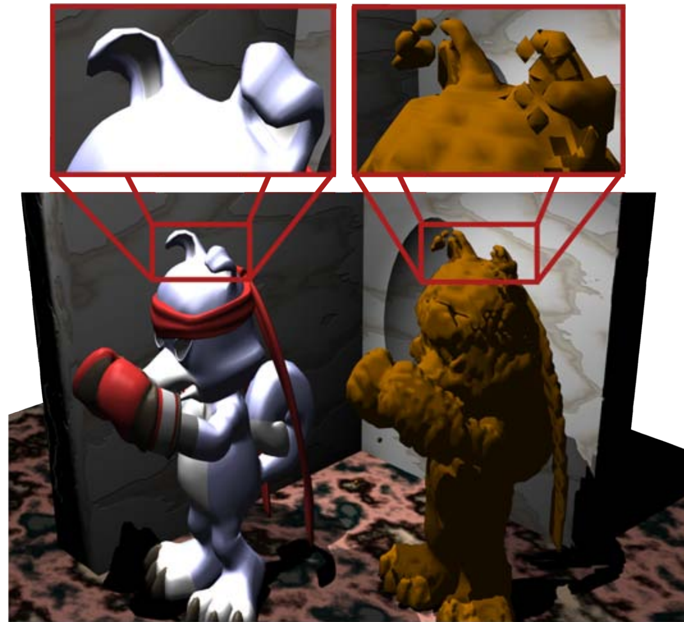


Figure 4.13: Artifacts due to the grid representation at a resolution of 80x100x80.
On the left is the gnomon creature model, as initially defined with curved surfaces. On the right is the corresponding representation of the model, after embedding into the LBM grid. The ears of the creature cannot be completely represented at this resolution.

University of Cape Town

4.5.7 Run-time vs grid resolution and code profiling

One of the objectives of this thesis is to provide estimates for the run time of fluid simulations for a specified level of detail. This is useful for visual effects teams, since they often need to estimate whether projects are feasible or if they need to simplify their fluid requirements for a given project. To this end an analysis of the run times of the single CPU implementation is provided. This will also aid the analysis of the multi-CPU implementation, as comparisons can be drawn and insight gained into the correct parallelization method.

To determine the resolutions for which it is feasible to produce simulations with a single CPU implementation, a simple experiment is conducted. Figure 4.14 shows the run times for the falling drop animation as a number of VOF LBM cells. The graph shows a strong linear relationship between the two variables. The computation time per cell from a linear regression analysis is $3.5 \times 10^{-4}s$, with an initialisation time of $1.44s$.

For a grid resolution of $300 \times 300 \times 300$, the predicted total computation time, for the 3.6s animation, would be $2.65hrs$. This is prohibitively large for such a short and simple simulation and would not be useful in a production environment. A reference animation of 1 min at this resolution would take 44 hours.

To profile the different elements of the simulation, Tuning and Analysis Utilities (TAU) [Mohr et al., 1994] was used to measure the time spent in different functions. Figure 4.15 gives the output summary, showing that the stream and collide steps do the majority of the computation in the simulation. The actual TAU output is given in Appendix C.1. Note that one of our performance optimizations was to move a large amount of computation from the collide function into the stream function to retain more cache coherency, causing the stream function to take most of the time. However, logically there is still a strict division between the stream and collide phases. We choose not to test this performance gain since this optimization is not new [Pohl and Rüde, 2007].

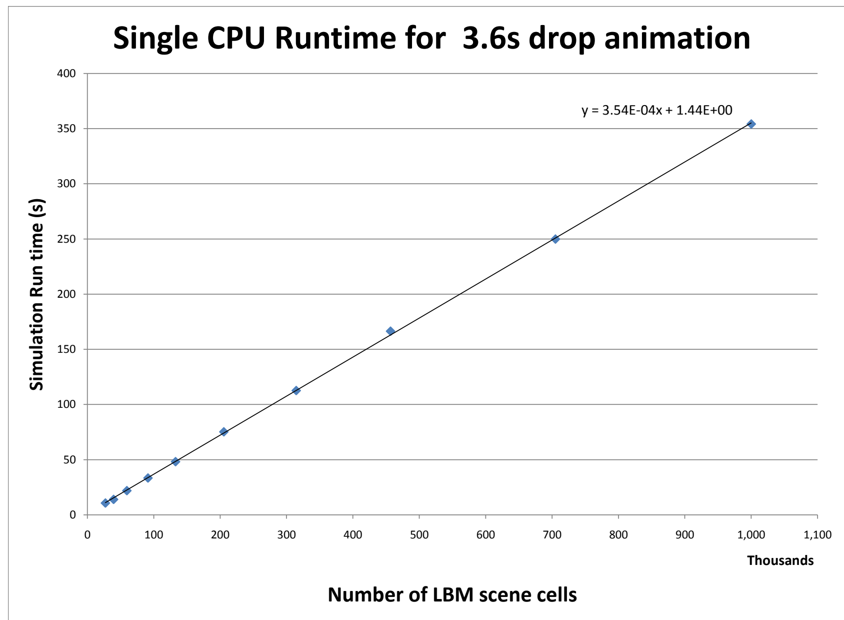


Figure 4.14: The time taken to produce the 3.6s falling drop animation for different resolutions.

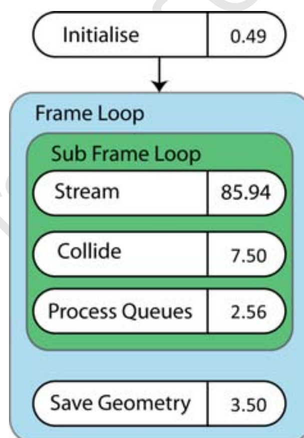
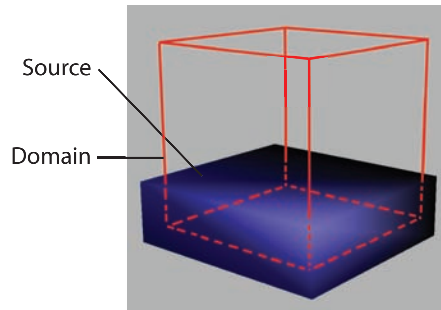


Figure 4.15: Percentage of time spent in each of the steps during a simulation.

Quantity	Type	Size(bytes)	Percentage
Mass	float	4	2.1
Pressure	float	4	2.1
Fluid Fraction	float	4	2.1
Velocity	float[3]	12	6.6
Distribution functions	$2 \times$ float[19]	152	83.1
Neighbour Counts	char[3]	3	1.6
Queue Position	int	4	2.1
TOTAL		183	100

Table 4.2: The space taken up by one cell in the VOF LBM implementation.



(a) Still pond

Figure 4.16: Still pond test case.

	Memory Accesses	Floating point Operations	Integer Operations
Operation Counts:			
Fluid Cell	536	401	144
Interface Cell	817	583	232
Percentage spreads:			
Fluid Cell			
Stream	6.62	0.00	0.00
Breakdown:			
DF propergation	100	100	100
Mass Tracking	0.00	0.00	0.00
Collide	93.38	100.00	100.00
Breakdown:			
Normal operations	37.50	23.94	56.94
Stability Correction	62.50	76.06	29.86
Interface Cell			
Stream	29.74	23.61	35.34
Breakdown:			
DF propergation	15.64	0.00	0.00
Mass Tracking	84.36	100	100
Collide	66.34	75.28	62.07
Breakdown:			
Normal operations	same as for Fluid Cells		
Stability Correction			

Figure 4.17: A analysis of the operations per cell in the stream and collide steps for the pond test case. The *operation counts* section indicates the total number of operations for the given step, while the *percentage spreads* gives the percentage of operations from the sub-step shown. The breakdown subsections give the spreads of the operations within the subsection. This analysis is for the still pond test case, as the cell distribution is predictable, but is a good indication of where the LBM computation is performed. It is interesting to note the collide step forms the majority of computation and the Smagorinsky stability forms a large percentage of these operations.

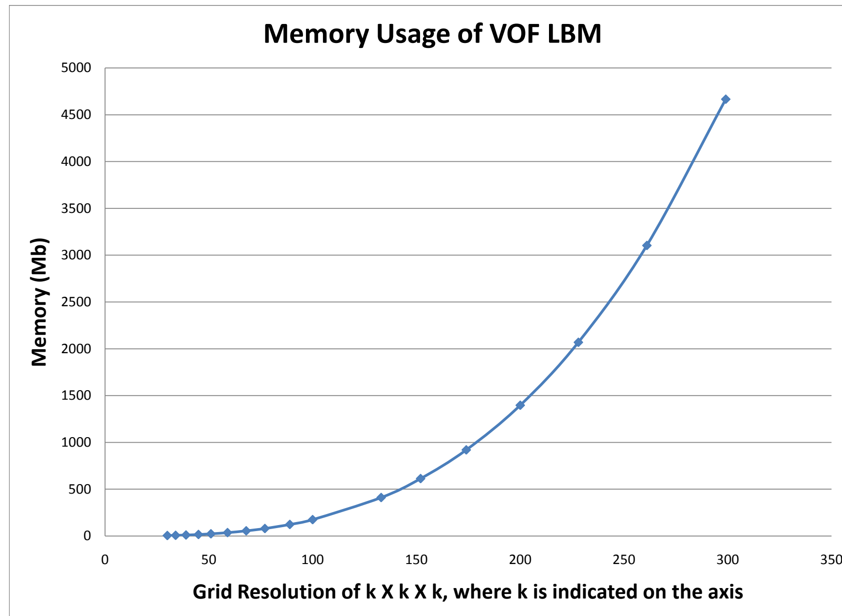


Figure 4.18: Memory usage of the VOF LBM implementation.
The graph shows that the memory usage becomes prohibitively large above a grid resolution of $200 \times 200 \times 200$ cells.

University of Cape Town

Figure 4.17 shows a summary of the distribution of memory accesses, floating point and integer operations within the stream and collide functions for the test case illustrated in Figure 4.16. This test case is used as the fluid remains constant and is more easily analysed. The LBM has strong reliance on memory accesses and floating point operations. Notice that the interface cells have more memory accesses due to the VOF mass tracking. Most of the simulation time is spent in the stream and collide steps, with the majority of operations in the collide step. This is especially true for the fluid cells, which will normally form the majority of the fluid in the scene unless the fluid in the scene consists of a large number of thin sheets. The Smagorinsky stability forms a large percentage of these operations (on average above 50% of the total cell operations for fluid cells and just under 50% for interface cells with respect to the stream and collide steps). The complexity of the VOF LBM is thus increased dramatically to main stability. A cheaper stabilization model would be a good area of further research to improve the computation efficiency of the VOF LBM algorithm.

Indeed, memory is a significant consideration of the LBM, as a large amount of storage space is needed per cell. Table 4.2 shows that 183 bytes of space are required per cell. Figure 4.18 shows the scaling of the memory usage with respect to the grid resolution and the memory used becomes very large above a resolution of $200 \times 200 \times 200$. To simulate large amounts of fluid, more then 4GB of memory is typically necessary.

Both the large amounts of memory required and the large run times for a simulation make the VOF LBM algorithm unattractive for use on a single PC. As most of the work done in the algorithm is in the stream and collide steps, both of which are per cell local operations, the algorithm would benefit considerably from parallelization. However, the large amount of data needed in the LBM might cause problems if the communication

between process or threads is slow.

4.6 Conclusion

The results have shown that the VOF LBM is a viable option for creating fluid simulations. However, it has not proven as robust as we hoped. Water behaviour is adequately captured by the method and by using smaller time steps, more turbulence can be captured (as shown in section 4.5.3). However, this cannot be an arbitrary choice as the simulation becomes unstable for very small time steps. The stability conditions for a given viscosity do not allow for much parameter tweaking, unless physically realistic results are required.

At lower grid resolutions the fluid surface is, in general, more bumpy, and the higher resolution smooths the bumps and gives more detail. Higher resolutions are also favourable for obstacle interaction, as more detail can then be resolved. However, they come at memory and computation costs (as shown in Section 4.5.4). The LBM needs significant amounts of memory for such resolutions and will benefit from a large cache. Appropriate cache optimisations are also advisable.

From the performance analysis in Section 4.5.7, it becomes clear that it is only attractive to simulate scenes with low resolutions on a single desktop. If larger scenes are required then using a cluster environment will be advantageous, as this will provide extra processing power and memory. In addition, to properly analyse the simulations possible with the VOF LBM, larger more complex scenes will be required. We perform this analysis in Chapter 6.

University of Cape Town

Chapter 5

Parallelization

University of Cape Town

5.1 Introduction

This chapter describes the conversion of our single CPU fluid simulation (detailed in Chapters 3 and 4) into a multi-CPU version. The intention is to reduce the computation time required to generate the simulations. Our target platform is be any form of computational cluster: a group of computers that are connected by a high speed network.

The single CPU simulation runs on the same computer as the interface (Houdini) and is tightly bound to Houdini as a plug-in. The multi-CPU version uses the same interface, but the simulation computation is farmed off to a back-end cluster by the plug-in, which generates and returns the fluid surface to Houdini. In simple terms, the back-end system uses multiple versions of the single CPU version running on each available CPU, with added logic for communication.

Load balancing is a problem for many physical simulations [Wilkinson and Allen, 1998]. The multi-CPU simulation should divide the workload up evenly among each of the processors to achieve an optimal load balance. For example, if we have 10 units of work and 10 CPUs, each of the CPUs could be given one unit of work. Our problem, however, is not so clearly defined, as the work to be performed is a complete simulation. This is a repetitive process of updating each grid location in the domain for each time step. As explained in Section 3, this requires information from neighbouring cells and, hence, the assignment of grid locations to a pool of CPU's must be efficient. Finally, each CPU should have all the required information to update the assigned grid locations.

The problem is further complicated by the fact that each grid location does not require the same amount of time to update. Trivially, a fluid cell takes far longer to update than an empty cell, although the difference in time required between a fluid cell and interface cell is far less (see Section 4.5.7 for a comparison). Even if we could perfectly estimate the difference in update time, fluid simulation is dynamic in nature and we cannot be sure of the scene fluid division ahead of time. Hence, we make use of load balancing to improve the algorithm efficiency.

The first step is the design of the back-end system, which makes use of the popular Message Passing In-

terface (MPI) protocol [Gropp et al., 1999]. This runs the simulation and returns the results to Houdini. In Section 2.6 we discuss previous approaches and parallel concepts to gain insight into the design. Using these ideas and definitions, the design of the parallel LBM implementation for this thesis is presented in Section 5.2, while the specific implementation details are given in Section 5.3.

5.2 Design

5.2.1 Algorithm Classification

The LBM algorithm can be converted to a parallel algorithm in a coarse grained fashion by splitting up the stream and collide loops. Every cell can be updated independently from the previous time-step's data for the neighbouring cell. Hence, we expect $T_s \ll T_p$, and that the algorithm should parallelize well. There is a medium level of data dependency, since it is a cellular automata system and requires neighbouring cell information to update a grid cell. This means synchronisation will be needed for every iteration of the simulation, which impacts on performance. Indeed, as previously mentioned, the problem is one of deciding which CPU will process which grid cell. The only sequential operations are the division of a given scene into units that can be processed by each CPU and the collection of the resulting fluid mesh once a frame of simulation is complete. There are additional operations that have to be run by every slave process, namely new operations (communication not included), that each CPU will have to perform to synchronize with neighbouring nodes. This extra time is created by the need to post- and pre- process messages.

We now discuss possible decompositions of the problem.

University of Cape Town

5.2.2 Master/Slave Division and Problem Decomposition

We use the master/slave model for parallelization, with 1 master and N slaves. However, there are many possible master/slave configurations. The master will always assign work units to the slaves and collate results.

One possible model has the master transmit the data for a number of grid cells to a slave, the slave then updates those grid cells and returns the data. This approach allows each slave to request work when it is idle. It does not require slave synchronization (as each slave will receive a complete set of data required to update the cells) and will have inherent load balancing properties. With this model, the total update time for k grid cells for one iteration can be modelled as

$$2T_{startup} + 2k \times \{cell\ data\ size\}T_{data} + T_{cell\ update} \times k,$$

where $T_{startup}$ and T_{data} are as defined in Section 2.6.2. The transmission of the fluid geometry is not taken into account. $T_{cell\ update}$ is expected to be small and the time required to transmit the required cell data will most likely dominate the update time for k cells. Additionally, the synchronization of each slave with the master causes a potential bottleneck for the simulation. The master also needs enough memory to store the entire scene. An entire scene, which can easily take up 2GB or more memory, will be transmitted every iteration.

Alternatively, the scene can be distributively stored across the compute nodes. The master then assigns

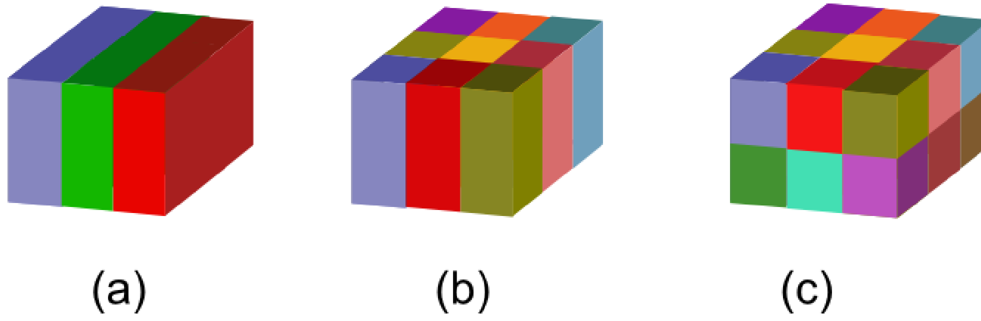


Figure 5.1: Domain decomposition methods.
(a) a 1D slice decomposition method along the x-axis, (c) a 2D box decomposition along the x and z axes and (c) a 3D cube decomposition along all three axes. Here the decomposition planes are across the entire domain, but they could divide up part of the domain unevenly, introducing more complex neighbour connectivity.

each grid cell to one of the slave processes at the beginning of the simulation. Each slave is responsible for updating the assigned grid cells and stores all the information necessary to update the grid cell. For the update, there is an important workload division: independent data versus dependent data (Figure 5.3). Independent data can be generated by the slaves at any time and is not dependent on a slave's neighbours, i.e. cells that do not share neighbours with cells assigned to other processes. Dependent data requires communication from neighbour slaves for every iteration, but a much smaller subset of the data is transmitted to synchronize slaves than in the previous model. Each slave no longer has to synchronize with the master.

University of Cape Town

Another possible model is to divide up the scene using domain partitioning (Figure 5.1). This is a simple method and shared boundaries can be regular. During the division, the ratio of independent data to dependent data is kept as high as possible by minimising the shared surface between neighbouring processes. Due to the grid structure of LBM, the simplest choice is to divide the scene in line with the axes. Figure 5.1 shows three possible axis based decompositions in one, two and three dimensions: the slice, box and cube methods. For our initial analysis, we consider decomposition planes that cut the entire domain, unlike the RCB method of Berger and Bokhari [1987], which uses variable splitting planes for each subdivision.

Our analysis of each of the three cases considers a grid size of $n \times n \times n$ completely filled with fluid and a pool of P slaves. For each decomposition, the initial distribution of the scene to a slave, assuming the slave is responsible for $\frac{n^3}{P}$ cells, would take

$$T_{startup} + \frac{n^3}{P} \times \{cell\ flag\ size\} \times T_{data}.$$

The variable $cell\ flag\ size$ is the size of the flag that denotes a fluid, interface, empty or obstacle cell. This distribution occurs at the start of the simulation and only the flag of the cell is sent. The initialisation time for the different decompositions will be similar, as the entire scene has to be distributed no matter what the decomposition.

All slaves must transmit synchronization data for each non-empty cell that is on the edge of its domain on every iteration, as the update of a cell is dependent on neighbouring non-empty cells. For the 1D slice case, this will performed with at most two other slaves and the communication time to synchronize slaves is

$$T_{startup} + n^2 x T_{data},$$

for each iteration, where x is the size of data that needs to be sent for correct synchronisation of a cell. The exact amount of data that needs to be synchronised will be defined in later sections. For the different decompositions, x remains constant and the value is not considered here. We assume here that data can be sent to more than one slave simultaneously (see section 5.3.3 for details on non-blocking communication). For this reason, we also neglect the synchronisation on the corners, as the bulk of the synchronisation will happen on flat surfaces.

For a 2D box division of u by v, where $u \times v = P$, with these being optimal integer values to maximize the number of independent cells, we get

$$T_{startup} + n^2 x T_{data} \times \frac{1}{v}.$$

For numbers which are not the product of two integers greater than one, the next closests nicely factorizable number can be chosen. For example if we wanted seven slaves, u and v would not be obvious and we then can have eight slaves instead. Further, without loss of generality, we assume $u > v$. In this case, each slave will need to synchronize with, at most, 8 other slaves, but 4 of the slaves will require far less communication, as they are only neighbours on a corner edge.

For 3D cubes, each slave will have to synchronize with, at most 18 others and the communication time will be

$$T_{startup} + n^2 x T_{data} \times \frac{1}{vw},$$

where the division is u by v by w and $u \times v \times w = P$ with $u > v > w$.

From this, we can see that of the three methods, the communication time for the 3D decomposition scales best, followed by the 2D and lastly the 1D case. However, this does not necessarily mean that these are the better options as they introduce added complexities and more operations are performed per iteration to manage the simulation. As mentioned previously, the workload of the simulation will not be evenly distributed and in the next section we will look at what effects this will have on our design.

5.2.3 Load Balancing

Thus far we have described a system where the simulation data is passed once from the master to a number of slaves that 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 one 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 make each slave run for the same time per iteration and, therefore, maximise efficiency.

Körner et al. [2006] suggest, but do not implement, a 1D load balancing scheme for the LBM. The scheme shifts the domain decomposition when a imbalance is detected. This is detected by measuring the wait time for synchronisations with neighbouring processes. When the wait time is overly large, a load balance is initiated and a plane (as it is a slice decomposition) of cells is fetched from the neighbour that causes the wait. This method works with all three of the domain decompositions and, thus, the load balance communication will scale the similarly to the synchronization communication. Although all cell data is transmitted during a load balancing phase, we can optimize the synchronization by only sending a subset of cell data.

University of Cape Town

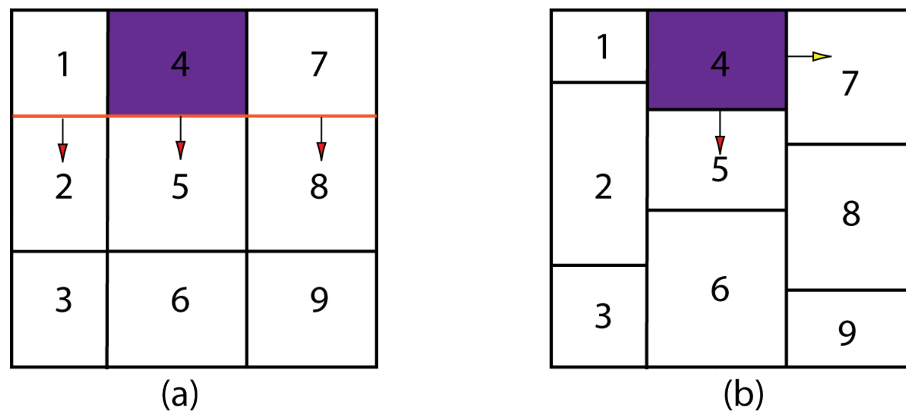


Figure 5.2: The 2D domain decomposition viewed from the top.
(a) A decomposition using planes that slice the entire domain keeping the division completely regular. Slave 4 detects an imbalance and moves its boundary downwards, but this causes a change in slave boundaries of 1,2,5,7 and 8. (b) Varied plane decomposition. The vertical planes cut entire domains, otherwise the decomposition will be completely irregular. Slave 4 detects an imbalance with 5, one boundary needs shifting between 4 and 5. In addition the boundary with 7 also needs shifting, in which case the boundaries of 4-9 will all be shifted.

University of Cape Town

The 2D and 3D domain decompositions scale better with a higher number of CPUs, but this does not necessarily make them the optimal choice. Consider the 2D case (Figure 5.2), where each slave has 4 possible neighbours to load balance against. The other 4 slaves on the corners are ignored, as they only share corner cells. A load balance consists of shifting the plane of the decomposition for a given slave into another slave's domain. In the previous section, we had planes that cut the entire domain as in (a) of Figure 5.2. So, when a plane is shifted by slave 4 towards 5, slaves 1, 7, 2 and 8 would need to load balance as well, which may be detrimental to the work division for those slaves.

On the other hand, we can have planes that cut the whole domain in one dimension and allow the other dimension to have irregular planes. There are two possible cases. In the first, when load balance occurs only two slaves will need to adjust their boundaries in a given direction (in Figure 5.2 (b), 4 shifts its boundary towards 5), but the slave-neighbour boundary conditions become complex. The other case, when 4 needs to shift its domain to the right, requires higher level logic, as the combined load of 4, 5 and 6 needs to be compared to the combined load of 7, 8 and 9. If only the load of 4 is considered, then again the overall load could be detrimentally effected. In the irregular plane case, it is possible that the neighbours which a slave synchronizes with may change, when the boundary between 4 and 5 shifts past the boundary between 7 and 8. The management of this design becomes increasingly unruly and difficult to implement effectively.

The 3D decomposition is similar to the 2D, but with additional complexities due to the extra dimension. The performance suffers further as gravity pulls the fluid to the bottom of the domain so that CPUs allocated to the upper sections will not have much fluid to update.

We choose the 1D decomposition for load balancing, as this keeps the load balancing logic simple and reduces effects due to multiple synchronizations 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. 1D decomposition also makes the code cleaner, as the slave-neighbours are always the same and have regular

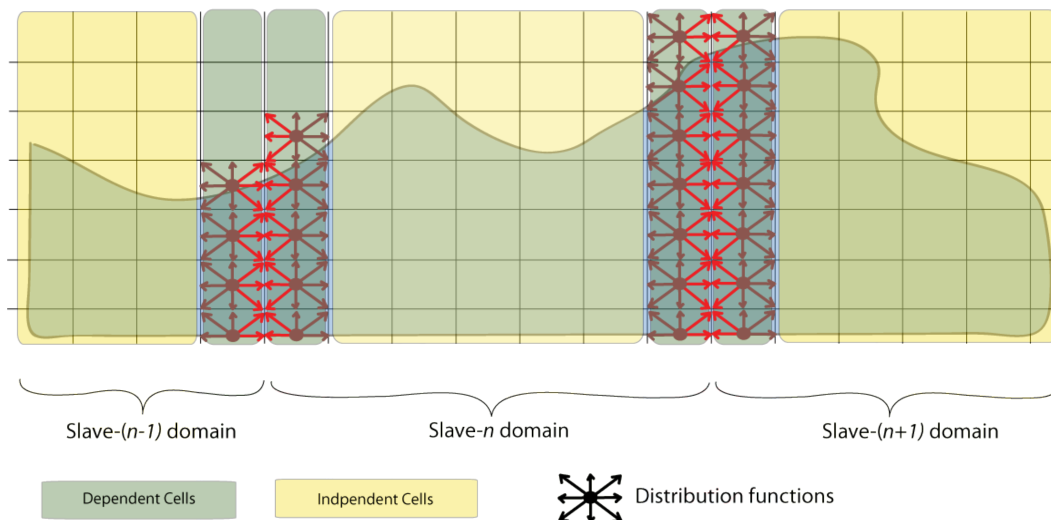


Figure 5.3: Dependent and independent cells.
Part of the 2D scene that has been divided up using the slice decomposition. Here, the slaves $n-1, n$ and $n+1$ are shown. The independent cells are in yellow on the outer edges of the domain, while the dependent cells are shown in green on the inside of the domain. The three DFs shown in red are required by the neighbour slaves and are packed together during a later synchronization. Other quantities required by the dependent cells are the mass, velocity, pressure.

University of Cape Town

plane boundaries. This is important, as the number of possible bugs is reduced and the performance analysis of the code is easier. The communication time does not scale as favourably as the box and cube cases, but, due to overlapping of computation and communication, we believe this will not be a problem.

The following section describes some of the finer details of the implementation of the multi-CPU fluid simulation and its load balancing.

5.3 Implementation

The description of the parallel algorithm starts with the basic workload division and an outline of the dependent data. We also cover the load balancing protocol and the technical challenges that arise, in particular with respect to HPC technologies.

5.3.1 1D domain decomposition

We use slice decomposition to divide up a scene into chunks of work for each CPU. The master node decides on the initial division of the scene by counting the fluid and interface cells, as this will account for most of the computation, and divides them as evenly as possible amongst the slaves. Figure 5.3 shows the division of part of a scene. The division is not perfect, as it is aligned with planes of the grid and each plane will not necessarily contain the same amount of fluid cells. A given slave, slave 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 for the cells for which it is responsible and two additional layers of *halo* cells (one above and one below), which hold the information received from a slave neighbour. The slave does no updating of the halos, but uses them to

look up values as normal 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 extended fill fractions, one additional plane of fill fractions beyond the halos, for correct normal calculations. These values are not needed for cells that are not near the fluid surface, so we reduce the amount of data that needs to be sent by only sending the cells that will be used in the surface construction. Namely, 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 $\beta < 0.5$. The appropriate data is then packed into a buffer and sent to slave neighbours.

5.3.2 Synchronisation

The fluid simulation generates distribution functions (DFs), pressure, mass and fill fractions. In addition there is the state of each cell given by the cell flag. The state shows whether a particular cell is either a fluid, gas, empty or obstacle cell.

Each slave is responsible for a part of the fluid domain and each of the above quantities and flag information requires synchronization. This is performed by sending some values to and receiving some values from the neighbouring slaves. Each send of data requires the use of external functions. Typically, these calls may require use of communication links such as an Ethernet network (see Section 5.3.5), that are relatively slow. Here, we outline our design to reduce the cost of these calls.

The simulation algorithm has a number of steps, as shown in Section 4.1. Each step requires some data from its own domain and others from neighbouring domains. Figure 5.4 indicates the data required and data updated in each step of the algorithm. Ideally, we want to be able to synchronize each of the updated data items after the appropriate step has finished. However, this would introduce a large communication overhead. Instead, steps are grouped according to what 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 of the local data. This is implemented using non-blocking communication with appropriate buffering (see Section 5.3.3).

Figure 5.4 shows that the Stream, Process Queues and Save geometry steps all require neighbour data and, hence, data needs to be synchronized three times per iteration. Process Queues adds additional complexities as the Filled and Emptied queues have to be processed in the correct order to avoid emptying cells adjacent to fluid cells erroneously. 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 then need to be informed of the change in the cells on the edge of the domain. To accomplish this, additional *filled halo* queues replicate the elements in the queue which are on the edge of domains so no further processing is necessary. In general, to maintain flag consistency across boundaries we do the following once a cell is detected as filled in the collide step:

```
if (fillFraction > 1.01*pressure)
{
```

University of Cape Town

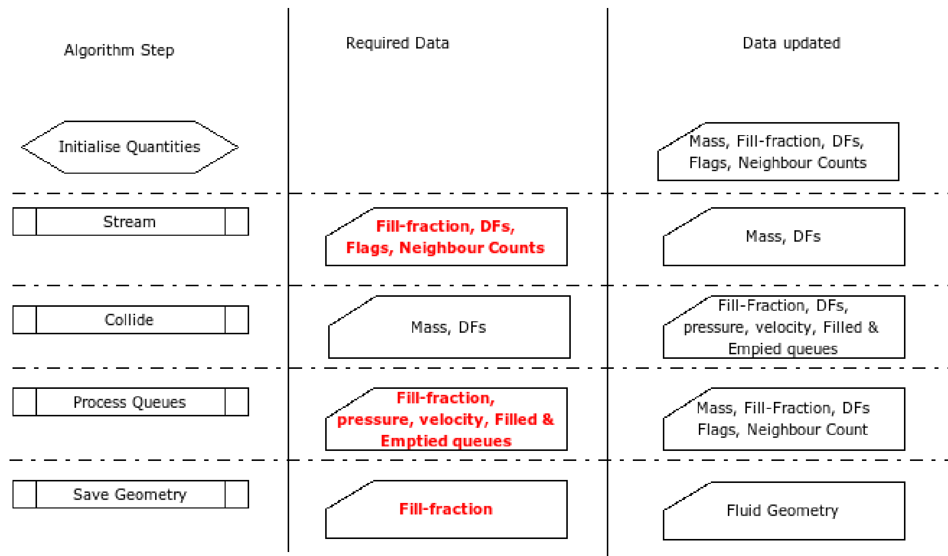


Figure 5.4: Simulation steps and required quantities.
This figure shows the individual steps of the sequential simulation and the required data for each step. Each steps operates on a cell and either only uses data contained in the *current cell* (shown in plain text) or from *neighbouring and current cells* (shown in red). The filled and emptied queues store which cells are changing from an interface cell to a fluid cell and an interface cell to an empty cell.

```
 for adj in adjacent cells
   if adj in emptiedQueue
     remove from queue

   mark adj as not_allowed_to_empty
 end

 if currentCellX == 0 or currentCellX == width−1 then
   add currentCell to filledHaloQueue
}
```

University of Cape Town

This ensures that the information stored in *filled halo* queues will be correct even after the subsequent Process Queues step is performed.

The *emptied halo* queue is formed by a post process of cells in the $x = 1$ and $x = width - 1$ plane, to see which ones are marked as emptied. The two queues can now be sent to their neighbours, as they are consistent and correctly ordered, and thus usable in updating their domain. The halo queues do not, however, tell the slaves when an interface cell has been added to the halo plane, which causes inconsistencies in the cell neighbour counts. An additional operation is performed to recount these values.

The two points of synchronization, Sync-A and Sync-B, are shown in Figure 5.5, with a small step re-ordering. These are before the stream step and after the collide step, respectively. The fill fraction is 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 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

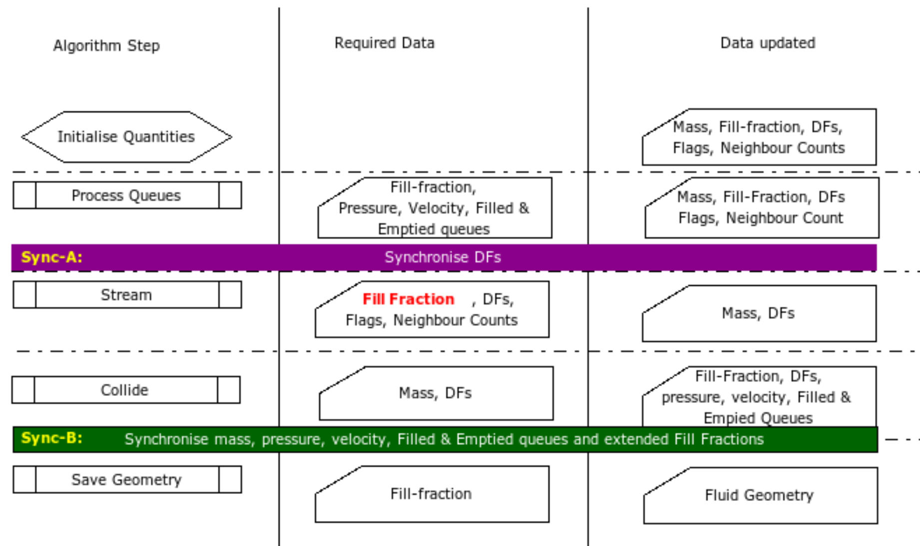


Figure 5.5: Simulation steps and required quantities with synchronisation. Figure 5.4 with the two points of synchronization and a re-ordering of steps. The fill fraction input into the Stream step has minor differences due to the Process Flag Changes step where mass is only distributed to cells within a slaves domain. The extended fill fractions are needed for normal calculations.

simulation is negligible. 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.

Using only two synchronisations steps optimizes the algorithm, but introduces error. To remove the possible error due to ignoring the effect of the process queues on the mass, an extra synchronisation step would be required. This would be infavourable for scaling.

In Sync-B the mass, pressure, velocity, filled queues, emptied queues and extended fill fractions are synchronized after the collide step. As it stands, with the Stream, Collide and Process Queues ordering, we found minor inconsistencies in the fill fractions while saving the geometry. This is undesirable, as there will be breaks in the fluid surface between slave geometries. To remedy this we note that the three steps are cyclic. Thus, if we re-order the steps so that the fluid fractions are synchronized before we save the geometry, then the final ordering is: Process Queues, Stream and Collide, after which the geometry is saved. 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). Since the simulation is non-deterministic this will not cause a problem, as the slight behaviour difference is unlikely to be seen by the human eye.

The synchronization phases follow the same format, they both pack data into a buffer and send it to a neighbour. The neighbour then unpacks the data into its grid for further processing. In general, this process of packing and repacking data adds extra run time to the simulation, but we found that the reduced time required to send the data over a network outweighs the extra time taken to pack data. Furthermore, it helps to organise your simulation data in contiguous sections of memory to reduce the time required to pack the data into a buffer for a send. One special operation that is performed during the Process Sync-B operation is the counting of 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.

University of Cape Town

5.3.3 Latency Hiding

We modified the initial design (outlined above) to improve the efficiency of communication. Due to the high communication time and the ability of network devices to buffer data and use DMA transfers, we are able to initiate sending and receiving operations and then perform appropriate computation while we wait for the data from a neighbour to arrive. The two MPI functions that expose these asynchronous features are `MPI_Isend` and `MPI_Irecv`. Each of the routines, once called, performs a small amount of preparation to allow the sending or receiving of the requested data and return to the parent function from which it was called, without waiting for the operation on data to complete. The routines `MPI_Test` and `MPI_Wait` are later used to determine whether the operation has been completed.

The algorithm we use is illustrated by the flow chart in Figure 5.6. The concept behind this design is to make the data needed for synchronization ready as early as possible, and quickly initiate the send of this data to slave-neighbours.

As in Figure 5.5, there are two synchronization phases. Each sync must communicate with all slave-neighbours unless the slave is the first or last in the whole domain. We can further split up each of these phases into three sub-phases:

1. Initiate
2. *Wait-for-Sync* to complete
3. Process Received data

The *Initiate* sub-phase is responsible for setting up the appropriate buffers, packing the data and calling the `MPI_Isend` or `MPI_Irecv` routines. The *Wait-for-Sync* sub-phase uses either a while loop with `MPI_Test` or the `MPI_Wait` function. Lastly, the *Process* sub-phase unpacks data received from slave-neighbours and updates local data accordingly. A full synchronization occurs when all three of these sub-phases are completed in sequence.

Figure 5.6 shows the life cycle of a slave process. First, the slave process is spawned on a node and is initialized by sends from the master process. The master process only sends the cell information to each slave for valid cells within the slave's domain. Therefore, an initial full synchronization of the halos is required. We perform Sync-B twice, as the filled and emptied queues add new cells to the halos during the scene pre-process to ensure fluid cells are not adjacent to empty cells. The new cell's mass, pressure and velocity will consequently be initialized with the appropriate values by all the slaves for each new cell in their domain, but these values require synchronization in the next Sync-B operation. If this does not occur then there will be erroneous values and simulation errors from incorrect boundary values. Lastly, a Sync-A is used to prepare the DFs for the streaming operation.

Once initialisation has occurred, the main simulation loop proceeds. The filled and emptied queues are processed first (as mentioned in section 5.3.2). After this, Sync-A is initiated, as the halo values are ready to be sent, following which the independent planes in the domain are immediately processed. Upon completion of a plane, we check to see whether the Sync-A has been completed. If it has, we process Sync-A and immediately Stream and Collide the halos making their data ready to be sent to slave-neighbours and Sync-B is initiated.

University of Cape Town

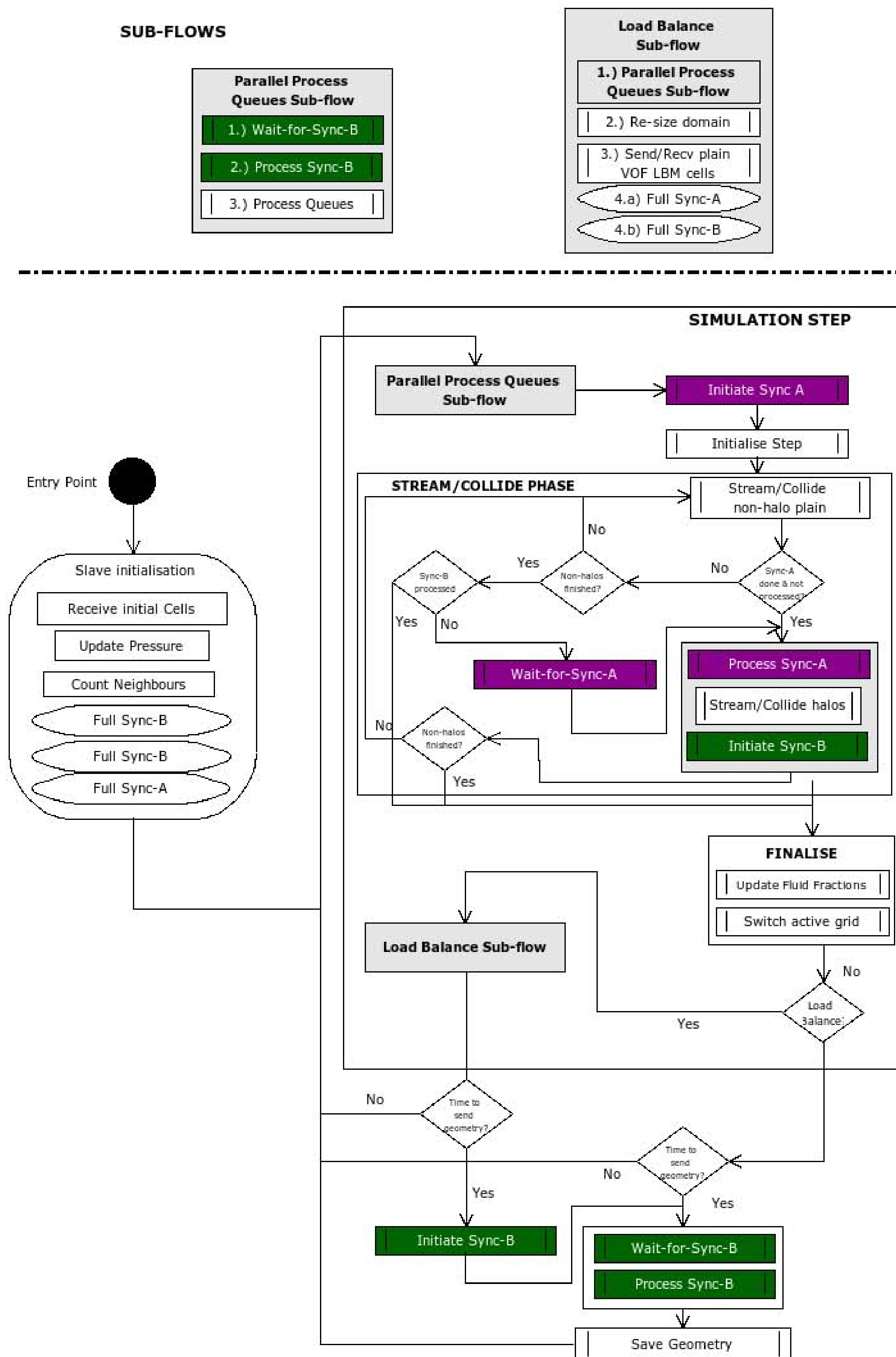


Figure 5.6: Flow diagram for each slave process.
The operational sub-flows illustrated at the top of the diagram are re-used later. Here, as much computation is overlapped with communication as possible. Each of the steps are denoted by rectangular boxed, with the sends being colour coded. The full synchronisations are given oval shapes as they perform the initiate, wait and process sub-steps, but do not perform direct computation on cells. Diamonds indicate decisions and control the flow of the program execution.

We then go back and finish the stream and collide operation on the independent data. Finally, after all the non-halo planes are completed we make sure that the Sync-B has completed and update the halos. Ideally, if there is perfect overlapping of computation and communication we will not have to wait at this point. However, this is unlikely and when the wait time is large, it is probable that load balancing is needed.

5.3.4 Load Balancing Protocol

The fluid will not always be evenly distributed across the domain and thus a static partitioning of the domain is not always an efficient strategy for partitioning computation. The dynamic strategy we apply, suggested by Körner et al. [2006], uses movable domain boundaries. We employ a slave-driven, idle process load balancing approach. Each slave has a set of slave-neighbours with which it synchronizes data. When a slave starts waits too long during either Sync-A or Sync-B, load balancing may be required.

The idle time period while waiting for each of the Sync operations to be finished, is stored separately for each slave-neighbour. If a wait time 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 balance phase from the slave-neighbour in question, by sending a message containing the time it has spent waiting. Upon receiving the request, the slave-neighbour will compare the received wait time to its current wait times and if it is appropriately larger, the request will be acknowledged. In this case, the two slaves go into a load balancing phase, during which the domains are re-sized. If however, the received wait time is not large enough, the load balancing is denied.

The load balancing phase proceeds as follows (see Figure 5.6):

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

The sending and receiving of any data during the load balance is implemented using blocking communication, as the domains are being restructured. The computation can only resume once the new domains are properly set-up. It is possible to use non-blocking communication as parts of the domain do remain constant, but this would require significant code enhancements, which we believe would outweigh the performance gains.

5.3.5 Technologies

MPI is language independent and has a number of specifications for different languages. We make use of the C and C++ specifications for MPI. There are many implementations of MPI for different platforms, but almost all high performance architectures have at least one. Each implementation has its own merits, some have been optimised for specific hardware (e.g., MVAPICH), while others are designed for multiple platforms (e.g., MPICH). Most of the implementations are very similar, as they follow the specifications closely, and the change from one to another, is simply a process of linking in the correct header files and libraries during compilation.

University of Cape Town

Cluster	Nodes	CPUs/Node	CPU Type	Interconnect	$T_{start-up}$	T_{data}	Memory/Node
CRUNCH	9	2	Dual core 2.4Ghz Xeon	Gigabit Ethernet	$1614\mu s$	$119Mbs^{-1}$	1GB
CHPC	160	4	Dual core 2.4Ghz Xeon	Infiniband	$87.06\mu s$	$396Mbs^{-1}$	17GB

Table 5.1: The two different clusters on which we ran our simulations.
We have direct use of the first system, while the second is a multi-user system.

We plan to run our MPI based simulations on two target platforms (Table 5.1):

1. A cluster of 64 bit Linux based servers (CHPC).
2. An Apple Macintosh cluster of Xservers (CRUNCH).

The CHPC is located off site and on a separate network, while the Apple cluster is on a local network. During our implementation of the back-end, we employed two different high level architecture designs. The first is a tightly coupled back-end that connects directly into Houdini, and the second is a loosely coupled architecture that exports a simulation from Houdini to disk, which is later run on the cluster, with the results being copied back.

The tightly coupled back-end is used in conjunction with the Apple cluster over a Virtual Private Network (VPN). The VPN is required, as the cluster is generally set-up such that one node is nominated as a head node, which is visible from the outside world, and in our case present on the local network. The head node and the other nodes in the cluster are connected to each other by a private network. MPI implementations, particularly those without added middleware software, such as GLOBUS in conjunction with MPICH-G2 [N.T., May 2003], require all nodes of a cluster to be on the same local network. A VPN allows a computer to join a private network, without it being physically attached to the network and have the same status as the other nodes on the private network. We used OpenVPN [Feilner, 1990] to allow one PC, the animator's PC running Houdini with the simulation plug-in, to join the Apple cluster, with Houdini itself running as an MPI process. The other MPI process are spawned on the cluster nodes.

Initially, we tried making use of OpenMPI [Gabriel et al., 2004] on the Apple Cluster, as this is an opensource MPI-2 implementation. Many parties were involved in its design and it is a new development, incorporating much from past MPI implementations. Therefore, it should perform well and have good usability. However, the VPN and OpenMPI do not work well together and Houdini was unable to communicate with the internal nodes due to the bridging of two network connections. OpenMPI operates at a low level on the network stack, and it tries to manage the network devices directly, bypassing the VPN translations. Instead, MPICH was used on the apple cluster, which allowed all the processes to see each other on the private network. One advantage of MPICH is that it has a high level implementation in Python, a scripting language, to control the management of processes, while the lower level network communications are optimised with a C implementation. We found the Python code useful, as we could debug problems with connections easily. With this combination of software, we were able to run tightly coupled simulations.

University of Cape Town

The VPN transports a message from Houdini to the correct MPI process in the cluster via the OpenVPN software link between the animators PC and the head node of the cluster. Naturally, this adds extra overhead. The local network connecting the animator's PC and the head node is a normal 100 Mbps Ethernet network, while the internal cluster network is Gigabit Ethernet. This does not pose a problem, as Houdini is on the master node, which is only used for initialisation, performed once, and receives the fluid mesh. The sending and receiving of the mesh can be perfectly overlapped with computation, as the fluid simulation does not rely on the mesh.

A loosely coupled back-end is required, as there is high security at the CHPC and establishing a VPN would not be possible. Additionally, while benchmarking the simulation, it is easier to run batch jobs from the command line using scripts rather then using the Houdini interface. The CHPC makes use of the IBM Load Leveler software, which is queuing software that allows many users to use a pool of CPUs. In total, there are 520 CPUs available for use at the CHPC, but there are many users who make use of these resources. During benchmarking, this will have to be taken into account, since in contrast, the results from the Apple cluster are for a single user system.

An additional complexity introduced by use of the CHPC is the 64-bit operating system. This is a necessity, as each node has 17GB RAM, which would not otherwise be accessible. The machine running Houdini had a 32 bit operating system installed and the saved simulations are therefore not directly compatible, due to two differences: (1) The type `size_t` is allocated 8 bytes on a 64 bit system, and only 4 bytes on a 32 bit system; (2) The size of pointers, and specifically the virtual address table pointers for classes, are now 8 bytes instead of 4. The other data types all retain the same size allocations. The advantage of access to large amounts memory is that compression techniques are not required.

The $T_{start-up}$ and T_{data} values are needed to estimate the communication cost and we us SKaMPI [Reussner, 2002] to obtain these times for each architecture. The values calculated are given in Table 5.1. Details of how these values were obtained are provided in Appendix D.1.

University of Cape Town

5.4 Conclusion

This chapter shows the conversion of the single CPU VOF LBM system into one which uses multiple CPUs. A simple 1D domain decomposition is used, as this simplifies the parallelization. The advantages are that the simplified version produces fewer bugs and adapts easily to dynamic load balancing. The more complex decompositions in two and three dimensions pose many problems when load balancing, as many slave domains change during a load balance. In one dimension only two slaves domains change during a load balance.

The 1D decomposition is at a disadvantage with respect to the scaling of communication time. In this case, the communication time remains constant for all numbers of CPUs and has lower predicted scaling when compared to the 2D decomposition. Overlapping computation and communication will help alleviate this problem.

The overall design of the algorithm uses one master and n slaves. The master loads a scene and assigns cell plains to slaves. The slaves then do the work of the simulation by running the VOF LBM for their domains. During each iteration of the algorithm, synchronisation is performed. In the case of dynamic load balancing, slaves monitor the time they spend waiting to synchronise with slave-neighbours for each iteration (although

this frequency can be varied). If this time is too large, then a load balance is ordered from the offending slave-neighbour. Once the required number of simulation steps have been performed, the resulting fluid mesh is returned the master process, which then saves it to disk.

In the following chapter, this implementation is analysed for efficiency and the general simulation quality produced.

Chapter 6

Evaluation

The aim of the implemented parallel VOF LBM is to produce fluid simulations, as efficiently as possible, with high levels of detail. This chapter evaluates the implementation for correctness and the speed with which different simulation scales can be produced. These form the main results of the thesis. The reader is reminded that simulation scales refer to how much fluid can be simulated, effectively the grid size used, as opposed to the spatial resolution, which measures distance between neighbouring cells. In the next chapter, the estimate of which simulation scales are feasible with the method and what architectures are most complementary will use the information presented here.

All the results in this chapter specifically test the implementation presented in Chapter 5. Section 6.1 provides the methodology for the results in Section 6.2, which is divided into: system scalability (a measurement of how well the system runs against the scene size and number of CPUs used), system correctness (do the fluid simulations generate correct fluid behaviour, using the single CPU implementation as a benchmark implementation) and a performance analysis (using a breakdown of the running time of each of the algorithm steps, this will indicate whether the algorithm is running correctly and why it exhibits certain behaviours).

University of Cape Town

6.1 Method

The system was tested against a number of test cases at different simulation scales. This is important, as the speed of the fluid simulation will be determined by a combination of the scene setup and simulation scale. The test cases used are based on the target application, namely animation sequences for movies and advertisements, as well as common tests cases employed in the literature. These test cases are (see Figure 6.1):

1. Water fall scene - Figure 6.1 (a) shows the curved river bed with a pump in upper left corner of the figure that fills the already half filled upper section of the river with water. The water then overflows from this point and falls to the lower level of the river. A sink is placed in the lower right of the scene to absorb the water.

2. Wave breaking over city - Figure 6.1 (c) and (d) show the scene setup and a perspective view of the city, respectively. To introduce water to the scene, a large block of water is placed in one corner of the

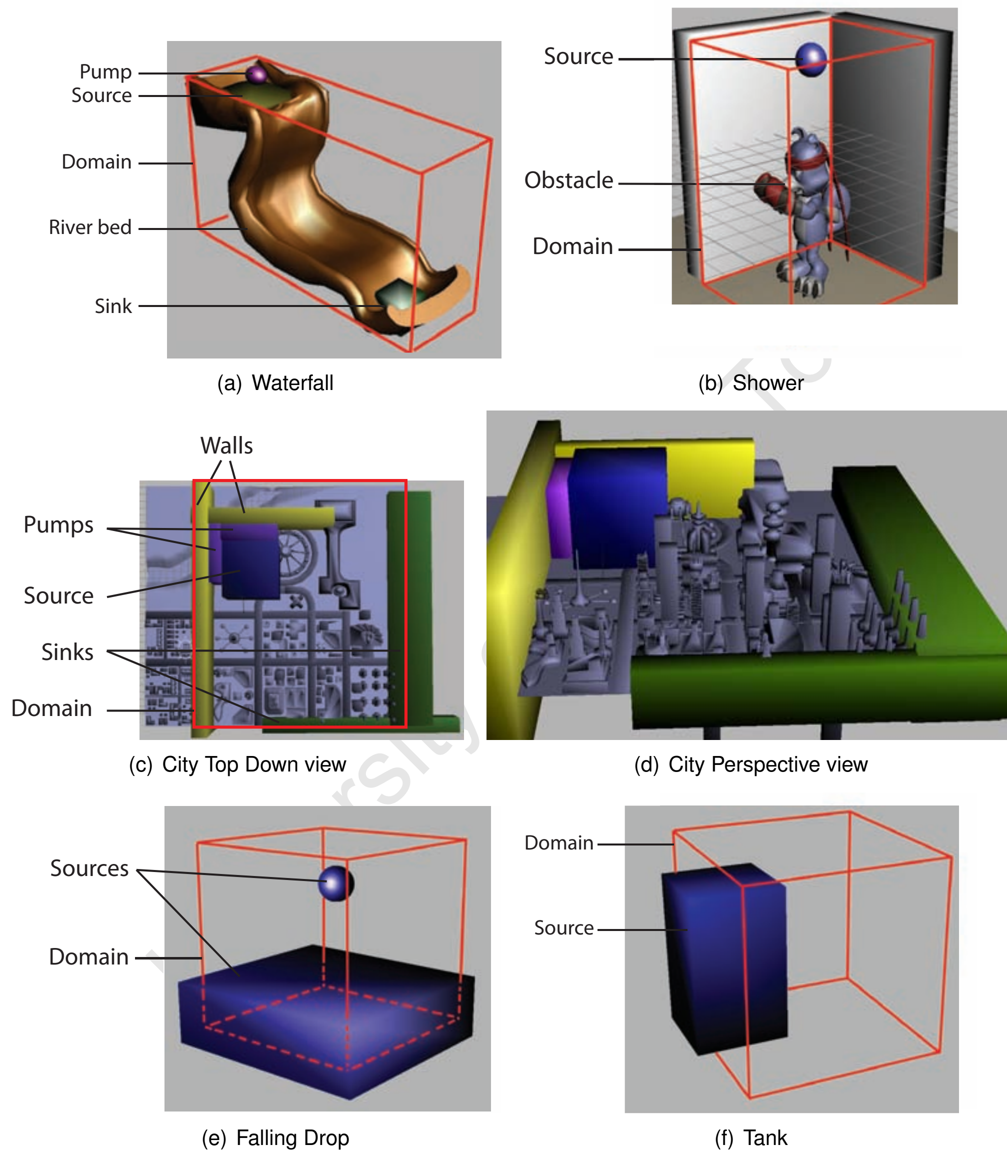


Figure 6.1: The five tests cases used to evaluate the parallel VOF LBM implementation. The scenes are constructed using obstacle, source, pump, sink and domain features.

domain, along with two pumps. The block provides the initial wave crashing on the city, while the pumps will continue to add water to the scene. Two obstacle walls, shown in yellow, are placed to guide the fluid in the correct direction and two sink walls (in green) on the other end of the city are used to absorb the water. This mimics the scene from *The Day After Tomorrow* where a city is flooded by a large wave[1].

3. Gnomon[2] showering in water/mud - Figure 6.1 (b) - depicts a creature being showered with fluid. Naturally, the fluid could be anything, such as water or mud. There is only one pump for the scene, a sphere above the creatures head. The inspiration for this scene is the film *Shrek*[3].

4. Breaking Dam - Figure 6.1 (f) is the standard falling dam from the literature [Thürey, 2003], with a block of water in the one corner of the domain. The water falls to the bottom of the domain under the effect of gravity.

5. Water drop falling into a pool - Figure 6.1 (e) is standard falling drop from literature [Pohl and Rüde, 2007]. The drop falls into the pool below.

To gain more insight into the algorithm's performance, the tests were run on two architectures (given in Table 5.1) to measure the impact of different technologies on simulation performance. As the cases and architectures are different, in the next sections, we use the definitions in Section 2.6.2 as normalized measures of the performance. The normalized measures can then be compared directly.

The following procedure was used to run each of the test simulations: The scene was specified in Houdini using geometry objects. These were processed and grid information was exported to a data file for a single CPU (it can obviously be exported for multiple CPUs, but due to the time required it was efficient to export just once). The file was then post-processed to create a grid file for a simulation using N slave processes. The grid file was then copied onto the cluster and the simulation run. The wall clock time was recorded from the time the master process started loading the grid file until it had saved the last frame's fluid mesh. These times are used to produce the information and graphs in this chapter.

University of Cape Town

6.1.1 System scalability

The system scalability was analysed using the definitions for wall clock time, scalability and efficiency provided in Section 2.6.2. It is also important to consider the wall clock time of the simulation with respect to turn around time in a production environment. From this measurement and the above scalability analysis, the highest simulation scale at which it is practical to simulate fluid can be determined. The scalability is a measure of how well the simulation is distributed over multiple CPUs.

6.1.2 Load Balancing

The multi-CPU version of the system has two implementations, one with static load balancing and another with dynamic load balancing. As the work load may not be evenly distributed across all the CPUs, the efficiency is reduced in certain cases. Load balancing seeks to improve the efficiency in these cases. Using the tools

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

Test	Architecture	Number of CPUs	Grid-size	LB	Computation Threshold	LB interval	LB Wait Threshold
case 1-5	CRUNCH	1-36	100^3	Yes/No	0-0.5	0-20	0-8
case 1-5	CHPC	1-512	100^3 - 600^3	Yes/No	0-0.5	0-20	0-8

Table 6.1: The tests that will be run to determine the scalability of the system.
The grid-size is an approximation as the scene may not have equal dimensions.

mentioned in Section 6.1.4, the amount of work performed by each processor and the time spent waiting for slave-neighbours will be recorded for both implementations to ascertain the effectiveness of the load balancing strategy.

Optimal parameters need to be selected to obtain the best results for the load balancing algorithm. There are 3 parameters: 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 wait time before load balancing is allowed. This seeks to avoid cascading load balances, as large wait times 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 wait time. A sparse set of tests will be used to determine the optimal values for these parameters for each architecture.

Once the optimal parameter values have been determined, the same tests are run for both multi-CPU implementations. Thus, a scene will be generated for each test case while varying the number of CPUs, simulation grid-size, as given in table 6.1. The results will then be compared.

6.1.3 System correctness

The correctness of the multi-CPU implementation is evaluated by a qualitative comparison, using the tool POLYMECO [Silva et al., 2005], of the meshes generated by the single and multi-CPU versions. This tool gives measurements, such as mean geometric and normal deviation of two meshes. However, even if the mesh is shown to be significantly different, the animation created may still be of sufficient quality, just with different behaviour.

University of Cape Town

6.1.4 Profiling

The Tuning and Analysis Utilities (TAU) [Mohr et al., 1994] are used to profile the system and provide an analysis of the run-time for each of the algorithm steps. This provides insight into why the algorithm scales as it does. The tool also provides a breakdown of which parts of the system are running for what lengths of time. By analysing the run-time of the stream/collide function, which is the workhorse of the simulation, we will be able to determine whether the load of the simulation has been balanced correctly. We will also be able to determine the overhead required by the communication for the parallel simulation and the load balancing.

6.2 Results

6.2.1 Animations

All the animations are rendered using Mantra[4] from SideFX, with various shaders being used for the different objects in the scene. The waterfall is shown in Figure 6.2, with the base of the river using a granite shader and the water using a basic fluid shader. The granite shader is a black stone, which sparkles in light reflection, and the basic fluid shader allows for the refraction and reflection, properties of fluid. In addition, the fluid has colour, which is more intense with depth. The fluid in the scene was simulated at a resolution of $500 \times 400 \times 200$.

Figure 6.3 shows the drop animation. The grid size for the final simulation was $600 \times 600 \times 600$. The floor of the scene is rendered with a wood shader, the fluid with the basic fluid shader and the walls with plain white shading. The drop appears to explode before it hits the surface due to the instabilities of the method.

Figure 6.4 demonstrates the shower animation using a clay shader on the fluid surface and two different marble shaders for the walls and floor. A spotlight has been added to the scene to add extra character. For this simulation a grid resolution of $480 \times 720 \times 480$ was used.

The city animation is shown in Figure 6.5 at a resolution of $600 \times 165 \times 600$. No texture or shader is applied to the city, but an extra environmental light has been added to introduce ambient occlusion to the rendering.

Figure 6.6 shows the breaking dam, which has a few artifacts, the most significant being the slow drip of the fluid down the side of the wall. Since the focus of the thesis is on the efficient production of the mesh surface and not the final look, this has not been further investigated. The scene was simulated at a resolution of $600 \times 600 \times 600$.

6.2.2 Parameter tuning

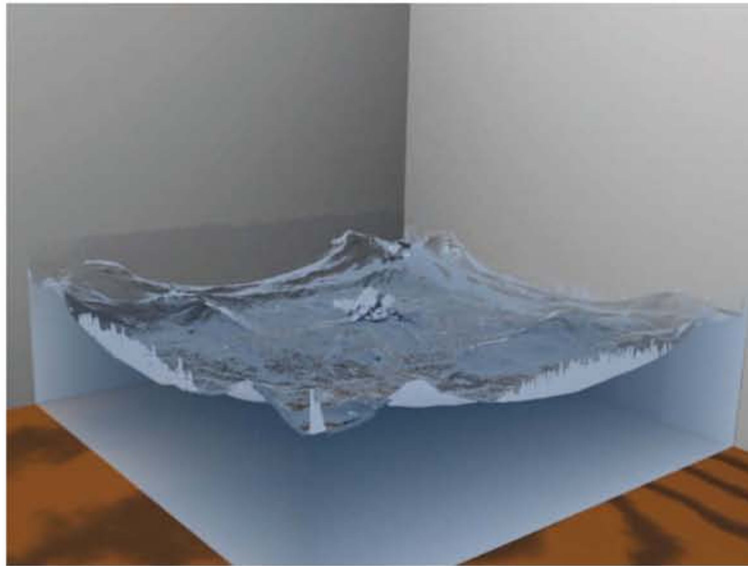
In general, an animator will not know how well the load will be balanced when specifying a scene in which fluid will be simulated. It is important to be able to choose parameters that will work for multiple scenes with varying loads. Thus, the optimal value for each of the parameters are evaluated for two distinct test cases: the breaking dam and water drop. The first has with an unbalanced load, while the second is reasonably well balanced load. Evaluating the performance for both cases gives an overall idea of how the parameters influence performance and values are then chosen to optimise both run-times.

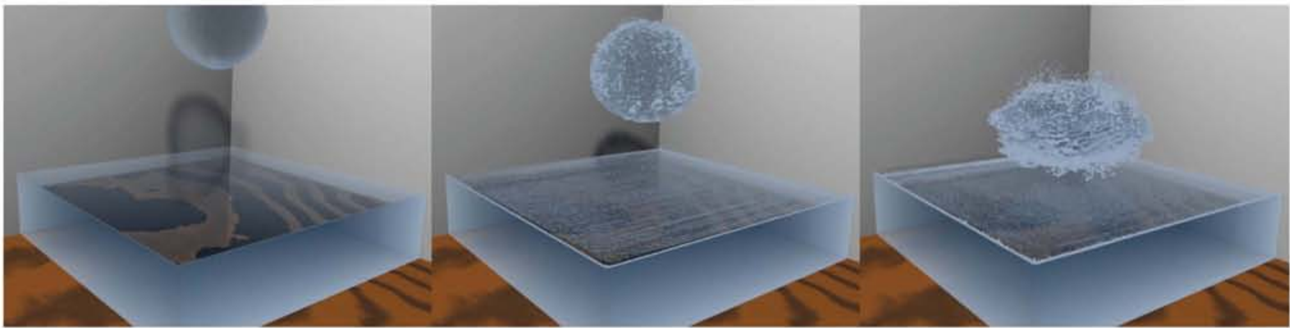
The optimal values for three parameters need to be found, which means running simulations for each possible combination of these parameters. We look at the change of two of the parameters at a time while the other is held constant. In addition, a fixed resolution is used for both cases, $100 \times 100 \times 100$ for the cases on CRUNCH and $200 \times 200 \times 200$ for those on CHPC. From the general trends, ranges and values can be selected for closer examination. The differences for the two architectures are given.

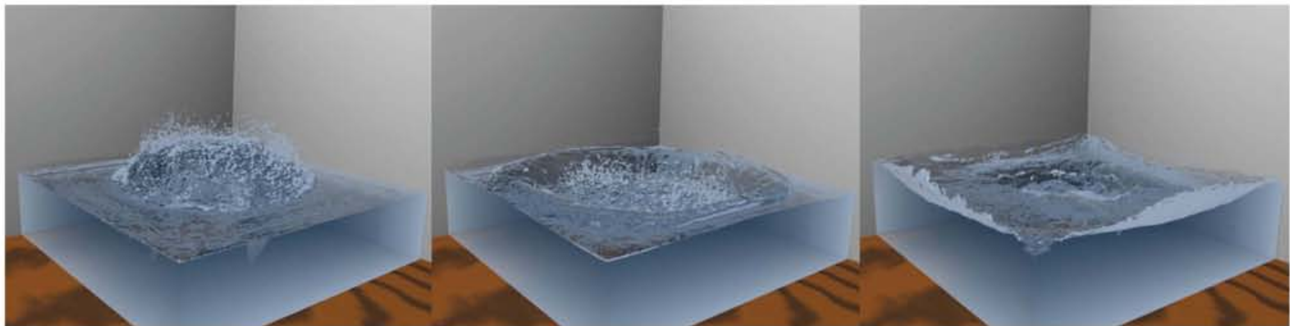
[4] www.sidefx.com

University of Cape Town

Figure 6.2: The final rendered version of the water fall scene.
The animation has a total of 150 frames. The last frame is shown large at the top of the figure, while the smaller images are every fifteenth frame of the animation, starting from the beginning of the animation.







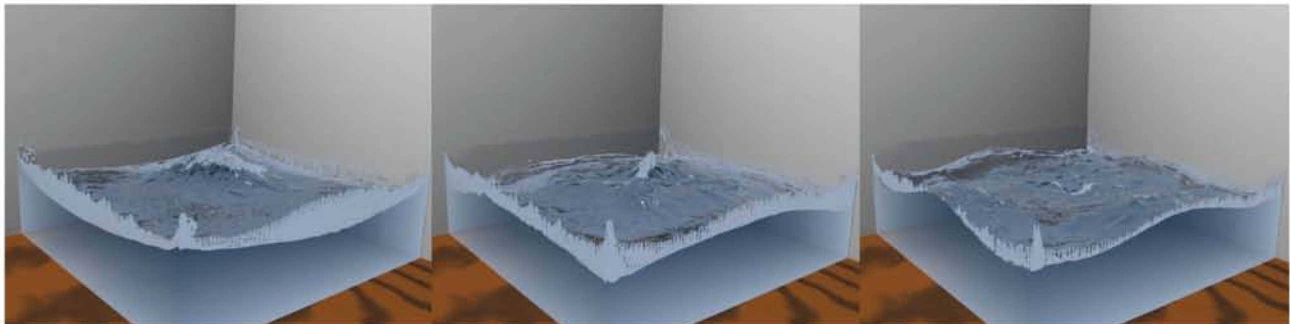


Figure 6.3: The final rendered version of the water drop falling into a pool.
The animation is a total of 72 frames of which 10 regularly sampled figures are shown.

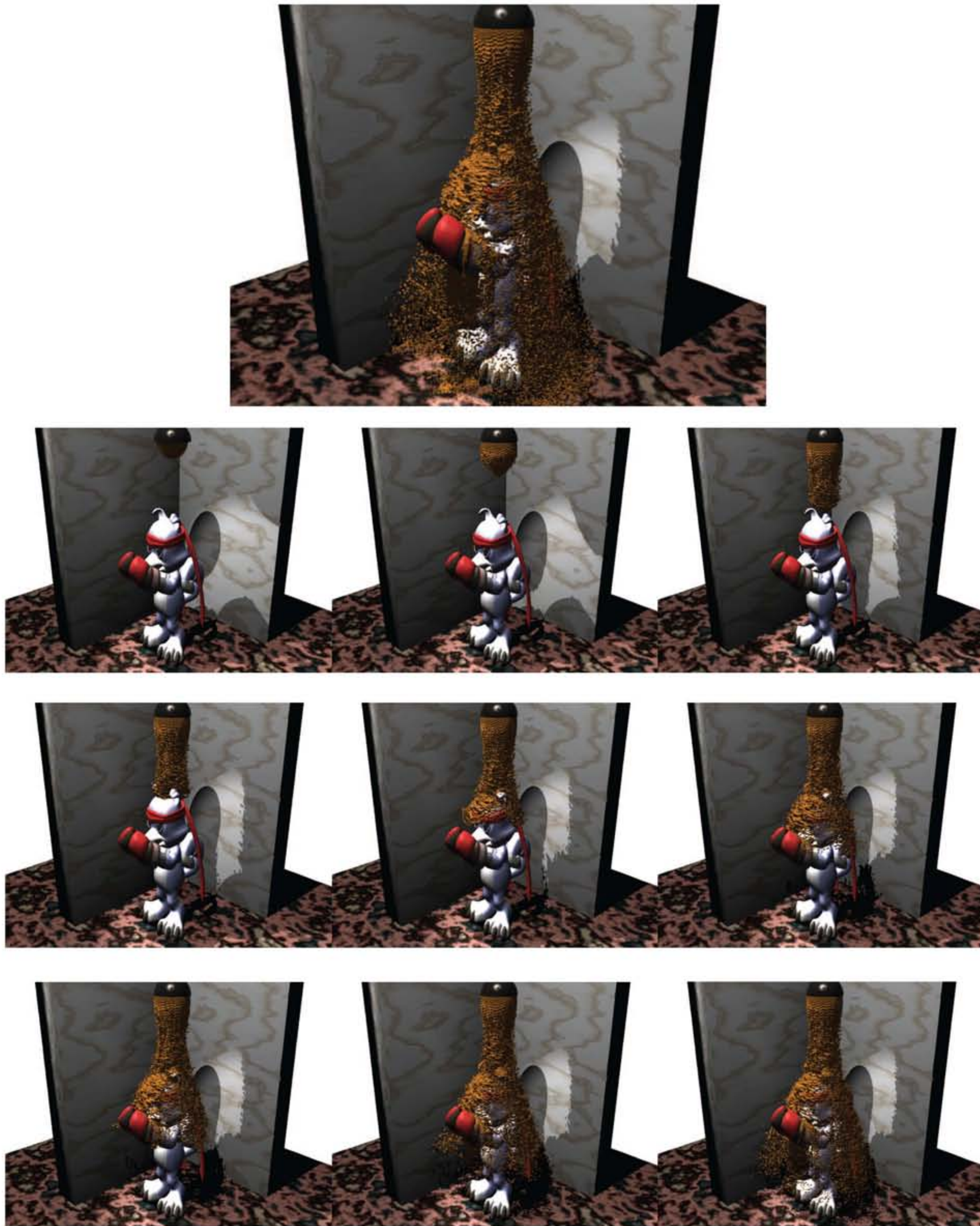


Figure 6.4: The final rendered version of the shower scene, with fluid rendered as mud. The animation has a total of 150 frames of which 10 regularly sampled figures are shown.

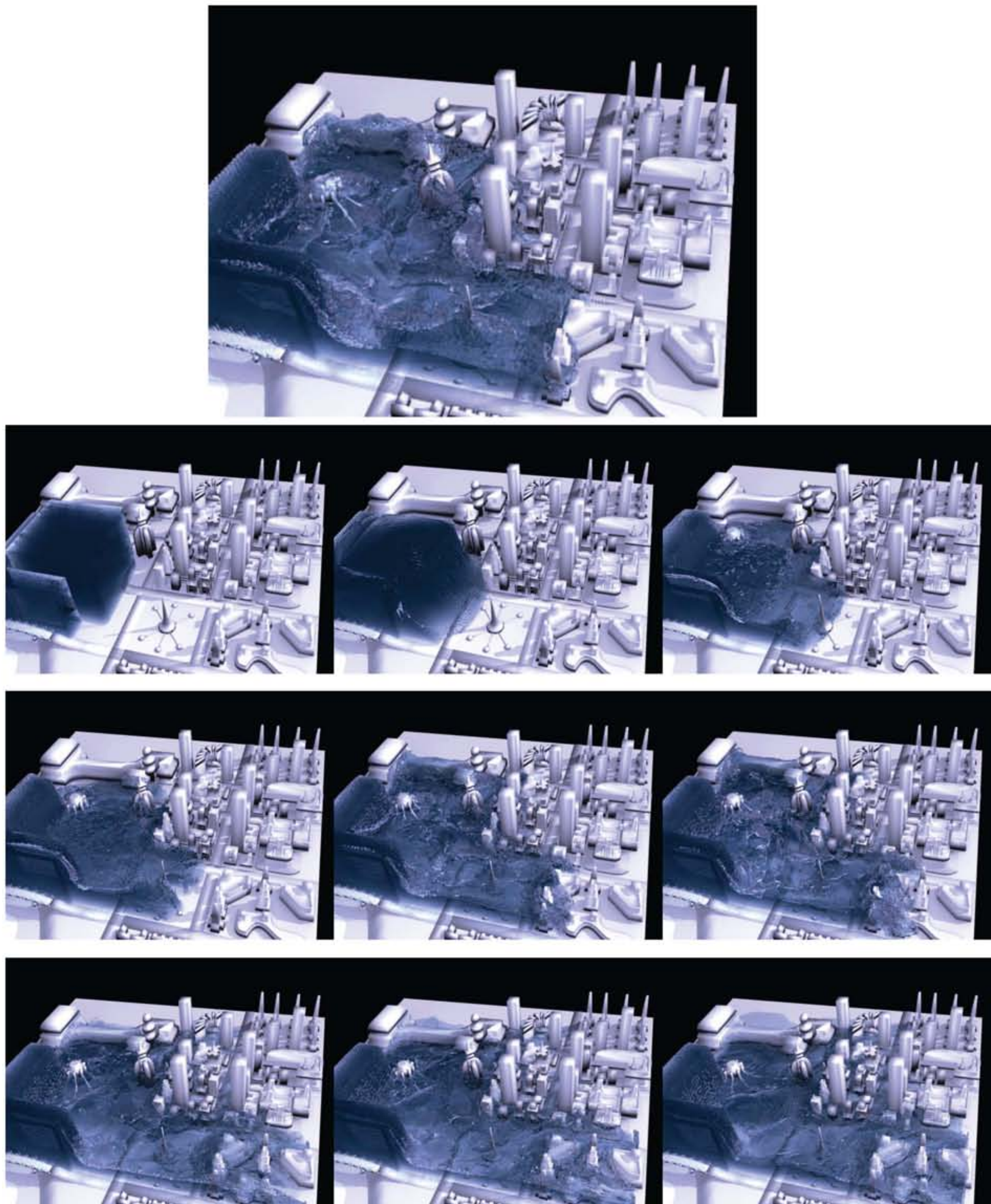
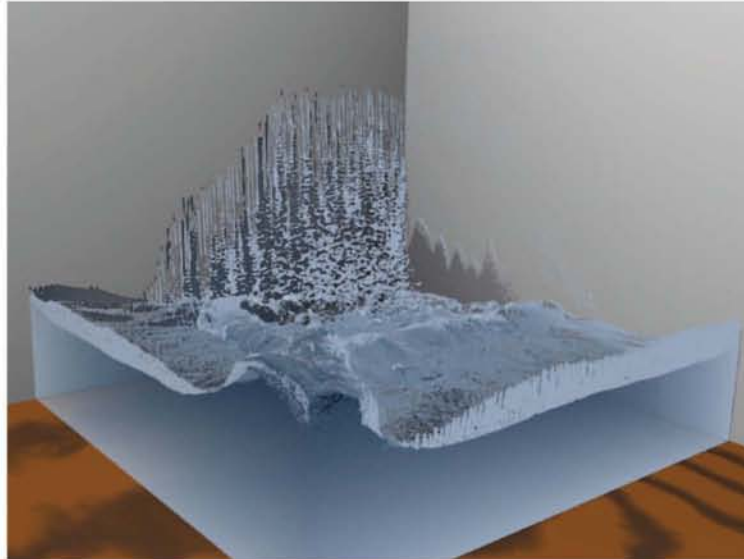


Figure 6.5: The final rendered version of the wave breaking over the city.
The animation is a total of 100 frames, of which 20 are shown.



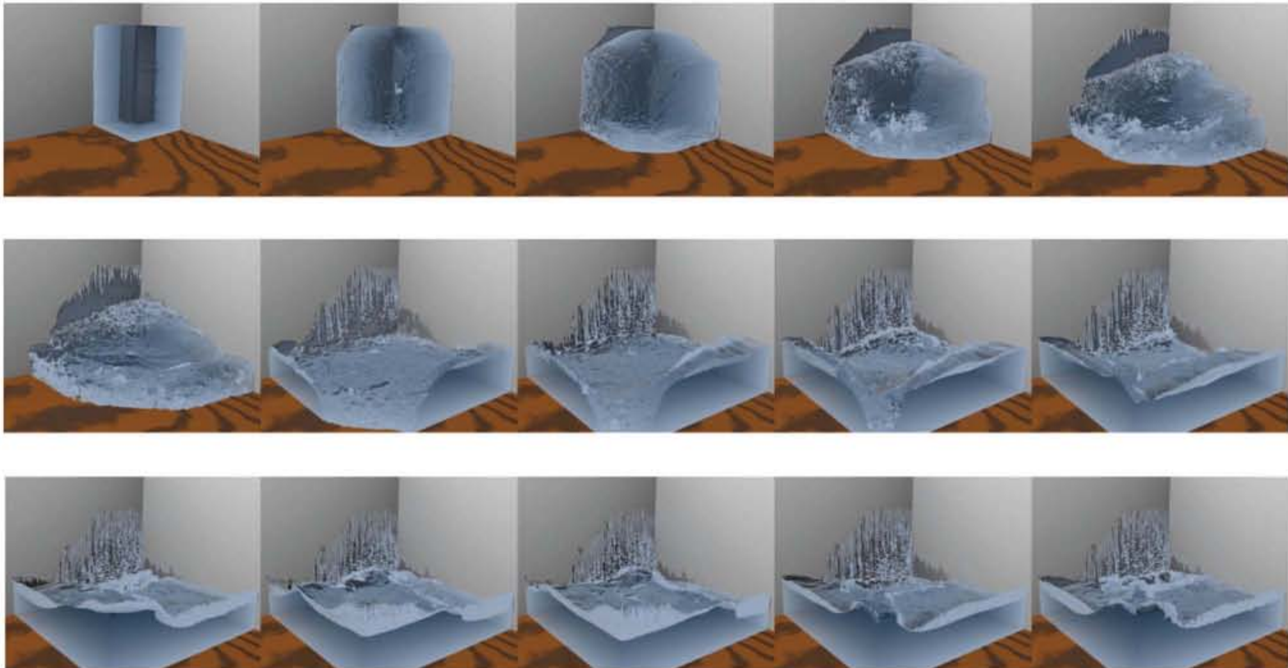


Figure 6.6: The final rendered version of the breaking dam.
The animation is a total of 72 frames of which 16 are shown.

6.2.2.1 CRUNCH

Figure 6.7 (a) shows the running time for the dam simulation, while varying the computation threshold (CT) in a large range from 0.025-0.375 and the wait factor (WF) from 0.5-3. The load balancing interval is kept at 1. In general, the running time is invariant under a changing wait factor between 0.5-3 and for computation thresholds between 0.025-0.275. For computation thresholds above 0.275, the running time increases dramatically. Figure 6.7 (b) explores a wait factor of 1-6 and a lower computation threshold range of 0.001-0.023. Again, the running time is largely invariant under the wait factor, while the run-time decreases as the computation threshold increases in the given range, until it reaches the plateau from the previous figure.

Figure 6.7 (c) explores larger wait factors for the dam simulation (1-71), which serve to slow the simulation down. Smaller wait factors are clearly more optimal. Figure 6.7 (d) shows the times of the simulation as the load balance interval changes (1-10), clearly showing that more frequent load balanced is considerably better.

Figure 6.8 shows similar graphs for the water drop simulation, but for slightly narrower regions of interest, as we want to find parameter values that are suitable for both simulations tested and use the better parameter regions found in the previous tests. The predominant trend with the drop test case is the less load balancing there is (high load balancing intervals), the better the run-time. No other significant trends were noticed, but performance changes do not seem as significant.

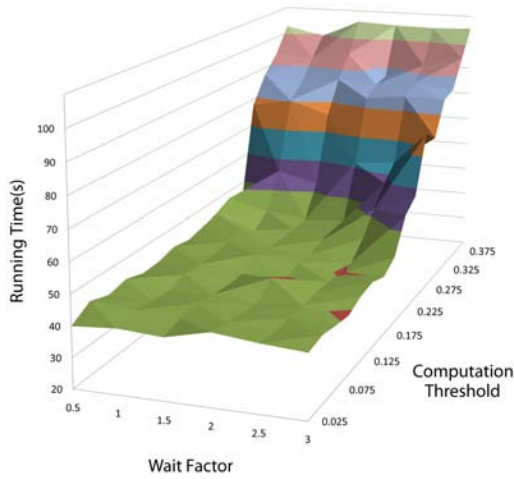
The invariance of the run-time due to wait factor value is worth discussing. The wait factor, w_f, allows a slave to reject a load balance request, which is necessary as the slave may be waiting for another slave to finish computation. Its value has little effect on the run-time, since slaves that have a high work load will have a very low maximum wait time, t_{max}, for neighbours. This wait time is multiplied by the wait factor, giving $w_f t_{max}$. This value is used to decide whether to reject a request that does not make it past the threshold. Since $t_{max} \ll 1$, so to will $w_f t_{max} \ll 1$. The value then becomes inconsequential in the decision process and w_f must be very large to influence the decision. Due to slow variance of the run-time with respect to the wait factor and this logic, we choose a value of 1, as this gives good results on the graphs and keeps the load balancing as sensitive as possible to changes.

University of Cape Town

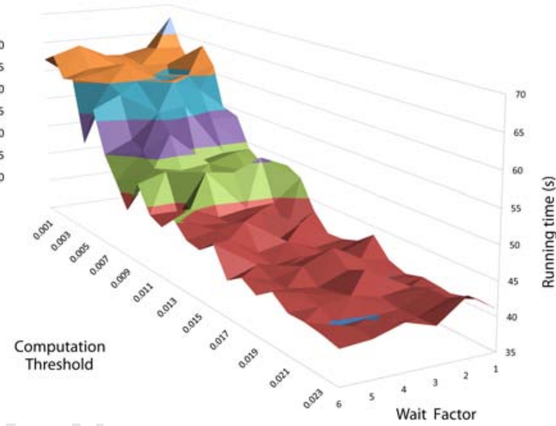
6.2.2.2 CHPC

As seen from the results on the CRUNCH architecture, the LB interval and the computation threshold play the biggest parts in the simulation times. The LB interval is chosen to be 1 to allow the system to adapt as quickly as possible to changes in the load. It is better if the system is able to make the correct decision more frequently. Additionally, it was shown that this gave optimal run-times for the unbalanced case of the breaking dam.

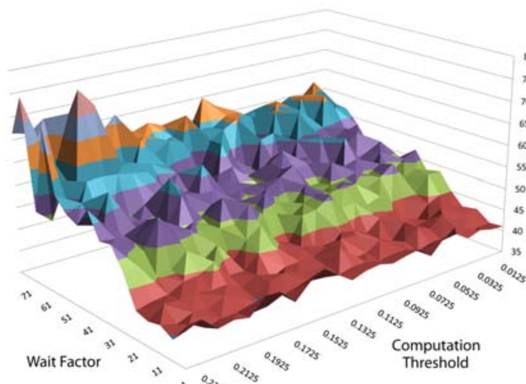
With the wait factor and the LB interval fixed, the breaking dam and the falling drop are run for different computation thresholds. Figure 6.9 shows the results. The drop simulation is better with no load balancing at all, while the dam has best run-times when the the load balancing is quite sensitive, using low values for the computation threshold. The optimal value for the threshold is not easy to choose, due to this contradiction.



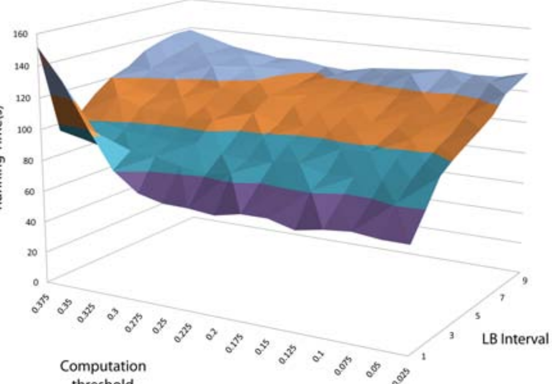
(a) WF=0.5-3.0, CT=0.025-0.375, LB Int =1



(b) WF=1-6, CT=0.001-0.023, LB Int =1



(c) WF=1-71, CT=0.0125-0.2325, LB Int = 1



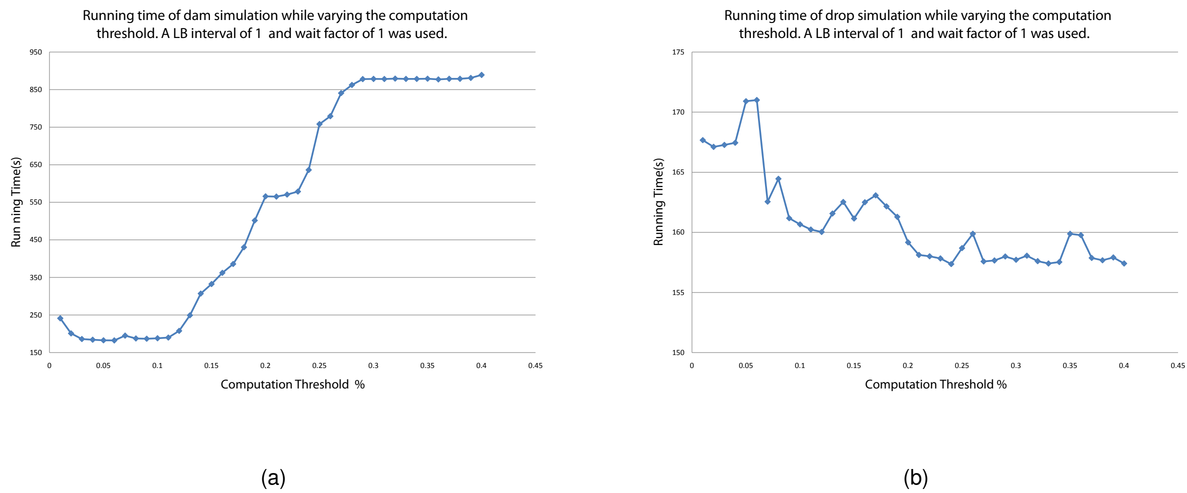
(d) LB Int=1-10, CT=0.025-0.375, WF=1

Figure 6.7: Dam simulation for different parameter values on CRUNCH.
The sharp rise in run-time for high computation thresholds and small LB intervals occurs as the slaves do not detect imbalanced loads with these parameters. Hence, the run-time is increased.

(a) WF=0.5-3.0, CT=0.025-0.375, LB Int =1

(b) LB Int=1-10, CT=0.025-0.375, WF =1

Figure 6.8: Drop simulation for different parameter values.



(a)

(b)

Figure 6.9: The run-time for the drop and dam simulations for different computation thresholds. (a) The dam simulation shows optimal run-times for low values of the computation threshold and increases for values above 0.1. LB rarely occurs above 0.3. (b) The drop simulation is better with no load balancing and the running time steadily decreases with higher computation thresholds.

6.2.3 Scalability and Efficiency

Here, the simulations are run for different numbers of CPUs (and the results shown as graphs in Figures 6.10 to 6.14). The simulation consists of running 1 master process and N slave processes. Note that for both CRUNCH and CHPC architectures, each compute node in the cluster 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 interconnect.

The CHPC is a multi-user system and the test results are effected by the load on the system at the time of running tests. This yielded variable results for similar runs of test cases. To alleviate this problem, each test on CHPC was run 5 times. The two largest times were discarded and the average of the three remaining times was used to produce the graphs. This goes some way to removing the variance due to high loads. CRUNCH had no such problems.

For the initial scene conditions that are loaded by the master process, there are two different creation methods. The first, for static load balancing, assigns a constant grid width to each of the slave processes. If the length of the grid along the x-axis is L and there are N slaves, then each slave gets assigned roughly $\frac{L}{N}$ slices. There are adjustments, of course, when L is not perfectly divisible by N. The second, for dynamic load balancing, assigns fluid cells evenly to all slaves. The total number of fluid cells in the scene is counted, let this be F, and then each slave is given as close to $\frac{F}{N}$ fluid cells as possible. Again the division is not perfect, as entire slices must be assigned to slaves.

University of Cape Town

6.2.3.1 Low Resolutions

The speedup and efficiency results for all the test cases, at a low grid resolution, are given in Figures 6.10 and 6.11 respectively. These results are from the CHPC platform, while the partial results for the CRUNCH platform are included in Appendix D.2.The first noticeable information from the graphs is that the simulations running on the CHPC architecture are markedly better than the CRUNCH architecture. There is little difference in the wall clock time for the single CPU implementation, with the CHPC being slower in both cases. The speedup and efficiency results show that the parallelization is not very efficient when more then 15 CPUs are used. At this point, the efficiency falls bellow 70% on CHPC. The dynamic load balancing algorithm performs marginally better than static load balancing up to around 17 CPUs, when the speedup is equal and then performs worse after this point. Although the difference in speedup is negligible overall. The graph of the dynamic load balancing algorithm is smooth, while the static load balancing increases in steps.

The performance difference for the drop and dam simulations with respect to static and dynamic load balancing is not very significant, but the dynamic load balancing is more efficient for lower number of CPUs on the CHPC. On CRUNCH, dynamic load balancing is in general worse, but remains within 5-10% of the efficiency of the static load balancing. For the CHPC, the performance of static load balancing eventually out performs the dynamic case, above 17 CPUs. A specific difference is noticed with the dynamic load balancing on the dam simulation, which has a 10% efficiency advantage for lower number of CPUs.

The mud and waterfall simulations are now considered. Both scenes are similar in nature, as water flows into part of the scene and fills the scene slowly. The graphs are then quite similar. Both do not scale well

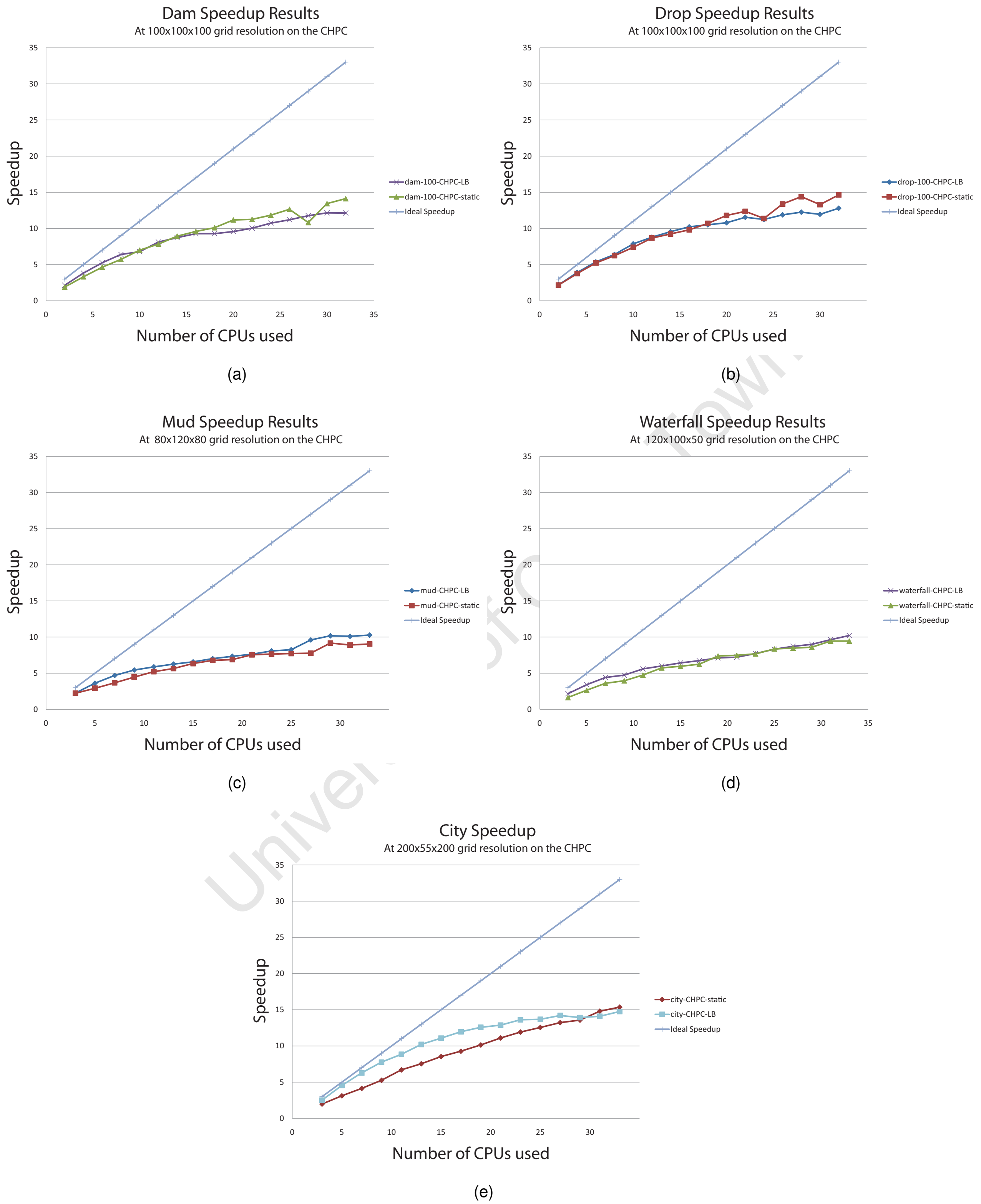
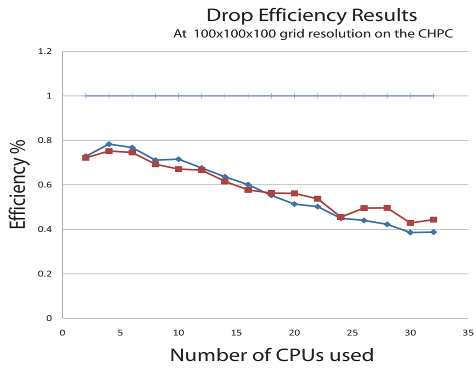
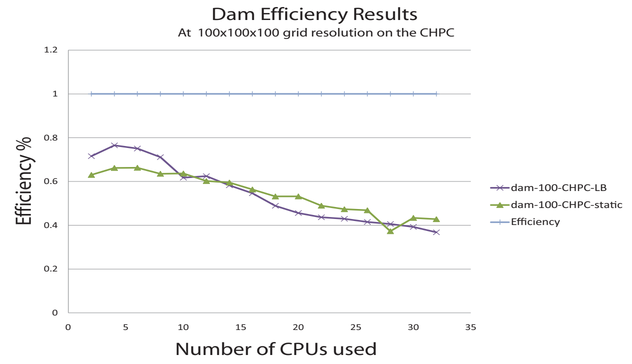


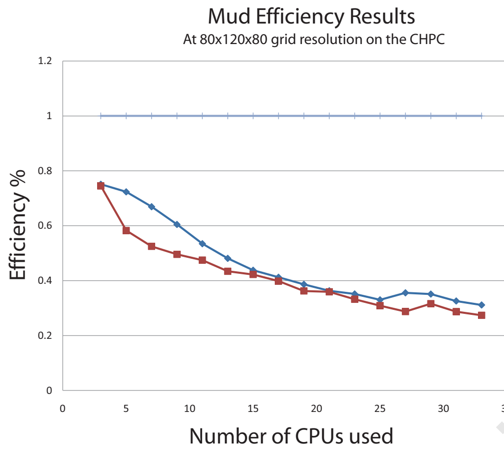
Figure 6.10: Speedup results for all test cases.



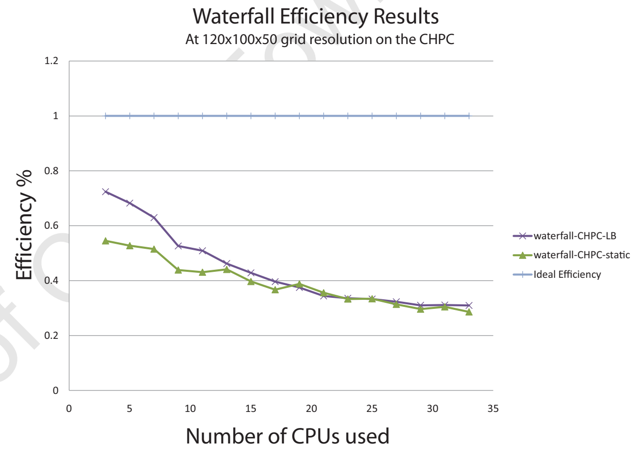
(a)



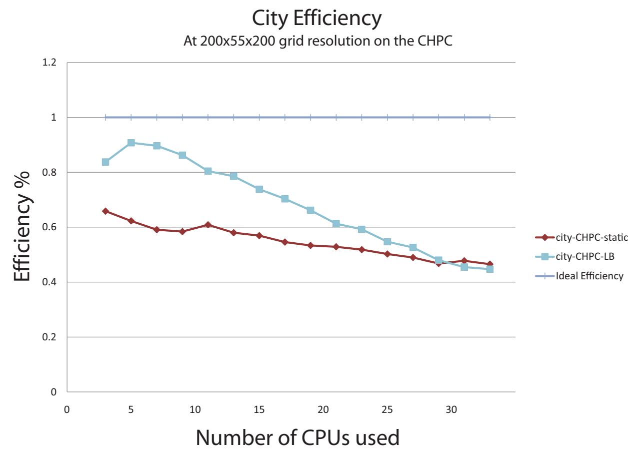
(b)



(c)



(d)



(e)

Figure 6.11: Efficiency results for low simulation resolutions.

above 10 CPUs. This can be seen most easily in the efficiency graph, which shows a drop of efficiency to below 50%. The performance of the static and dynamic load balancing are also both very similar, but dynamic load balancing gives about 3-20% efficiency improvement in the range of 5-12CPUs.

The city simulation results show a much larger performance difference between the dynamic and static load balancing. The dynamic load balancing achieves an efficiency above 90% for 5 CPUs. The crossover point for the static and dynamic cases is at 32 CPUs.

The wall clock times (see Appendix D.2) for all of the simulations are good and the simulations can all be generated in under 2 minutes with less then 10 CPUs. Another notable feature, present on some of the efficiency graphs for the CHPC, is that the efficiency increases from one node to two.

6.2.3.2 Medium resolutions

Here, we present the results from running the simulations at medium resolutions. Due to the memory limitations on the crunch architecture, the tests were only run on the CHPC. The wall clock times are included in the Appendix D.2, Figure D.7, as they are similar to the previous wall clock graphs and the speedup/efficiency graphs provide greater insight. In general, the wall clock times show a fast reduction in time when using up to 10-15 CPUs and then far less improved performance for higher CPUs.

The speedup and efficiency results for all the test cases, at a medium grid resolution, are given in Figures 6.15 and 6.14 respectively. The graphs of the drop and dam show some of the same characteristics as their low resolution counterparts. The performance stepping that is present for the static load balancing case is also present in the medium resolution cases. The dynamic load balancing shows a smooth graph, indicating that the load is more evenly balanced. However, again there is a crossover point where the static load balancing has better performance. For the drop simulation, this occurs at 20 CPUs and for the dam simulation between 15-20 CPUs (most easily seen on the efficiency graph). There is the same increase in efficiency from 2 to 10 CPUs and then a subsequent decrease.

The waterfall and mud results also exhibit similar properties. In general, poor scaling beyond 10 CPUs is observed and slightly better performance for dynamic load balancing until a crossover point. The city simulation shows a far larger efficiency gain with the dynamic load balancing, but ultimately poor scaling beyond 30 CPUs.

When comparing the low resolution results against the medium resolution results, the drop and dam simulations evidence better efficiency and, therefore, better scaling. The mud and water simulations have similar efficiency results and the city poorer results.

6.2.3.3 High Resolutions

The speedup and efficiency results for all the test cases, at a high grid resolution, are given in Figures 6.10 and 6.11 respectively. As the memory required to run a simulation on a single CPU is prohibitively large, an estimate for the single CPU run-time is obtained by fitting a power regression line to the static load balancing data. This value is then used in the calculation of the speedup and efficiency.

University of Cape Town

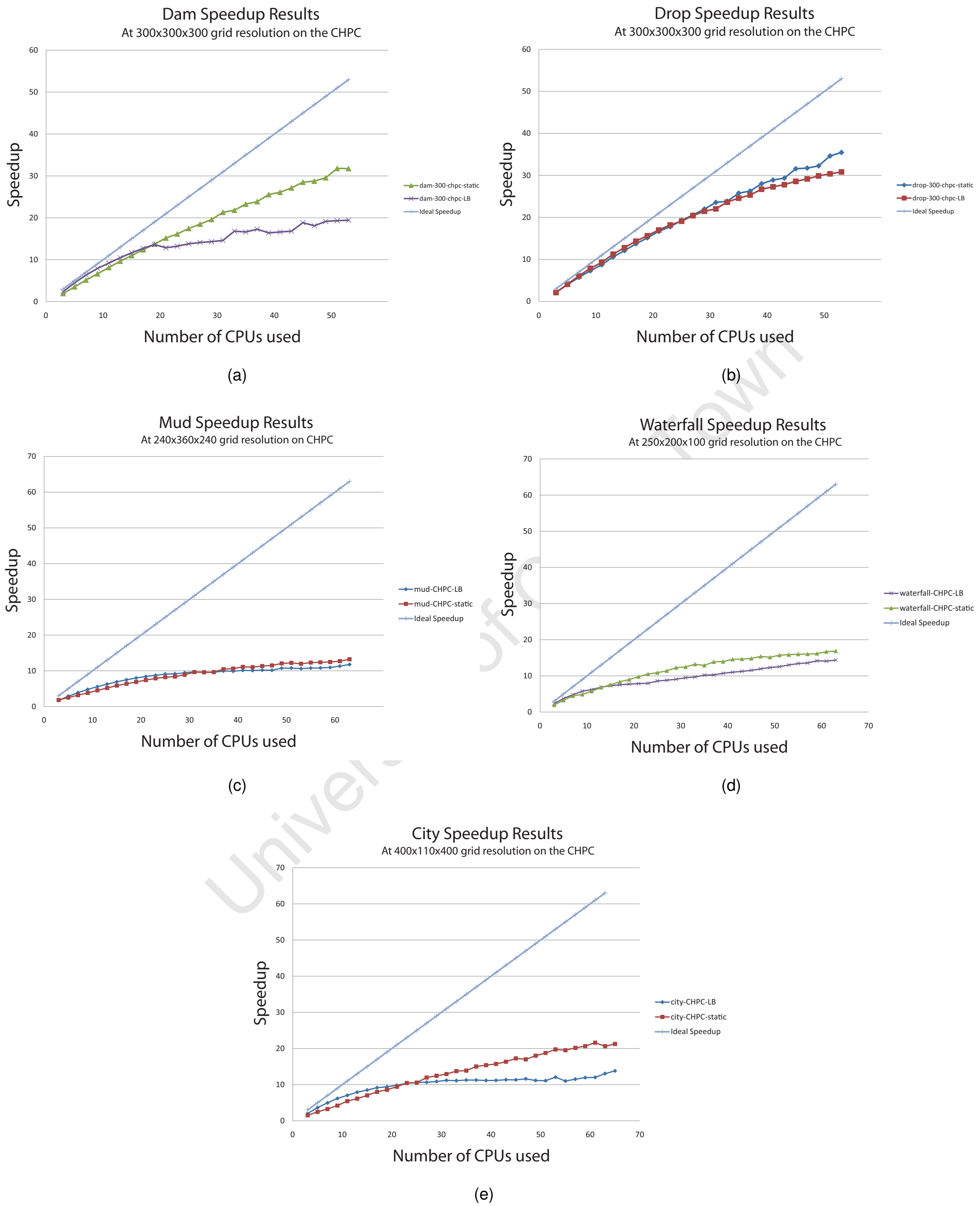
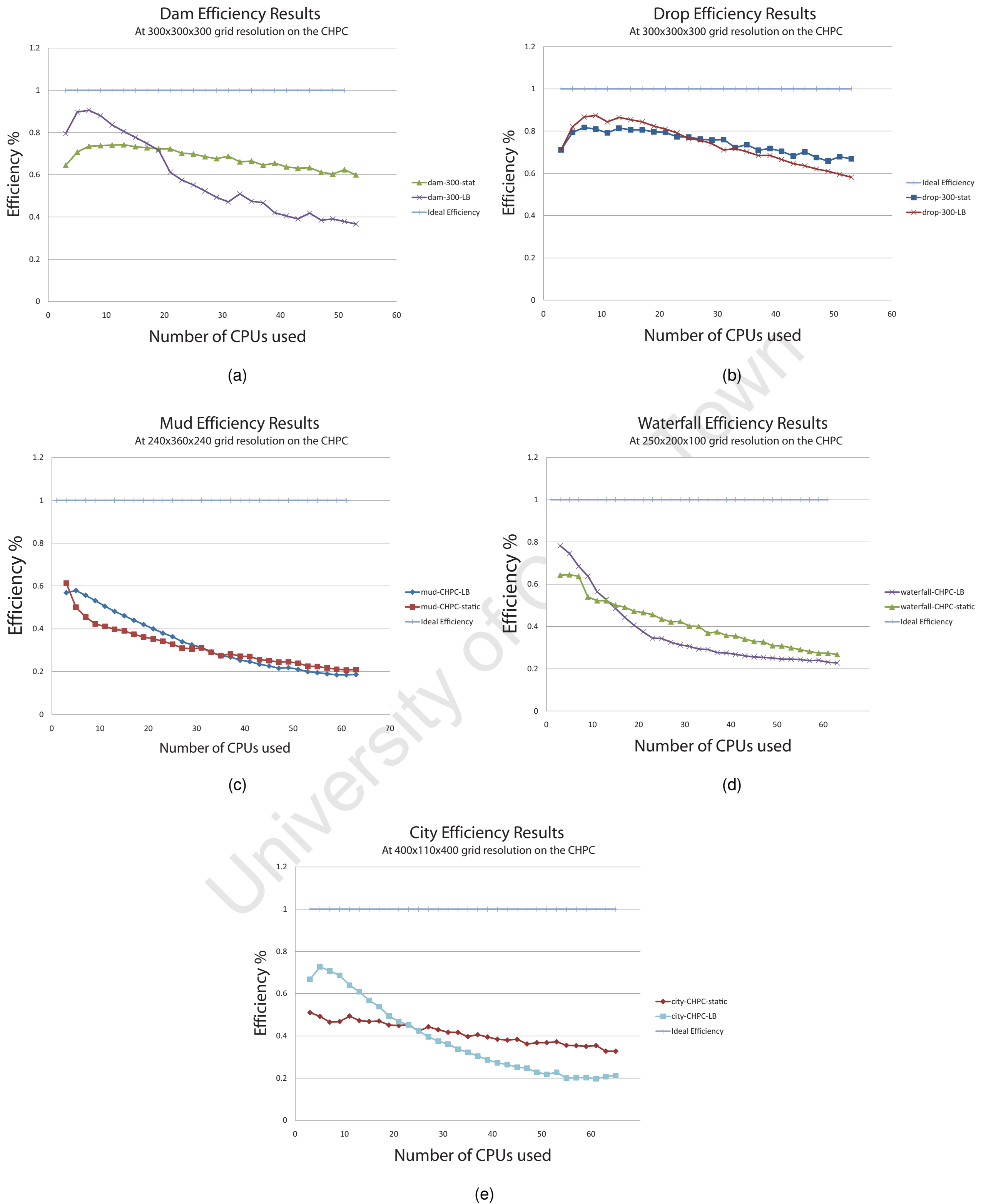


Figure 6.12: The speedup results for the test cases at a medium resolution..



University of Cape Town

Figure 6.13: The efficiency results for the test cases at a medium resolution.

Note for the dam and waterfall simulations at a high resolution the algorithm runs into memory limitations. This is due to the initial division of the scene for the load balancing simulations (refer to the introduction of Section 6.2.3), which allocates a large empty portion of the scene to one slave and virtual memory is used. For the dynamic dam simulation, the results included here use the same starting conditions as the static case. This is why the graph appears as presented, with very bad results in the beginning and then slowly matches the results of the static load balancing after some time.

In general, the graphs have similar characteristics to the graphs from the lower resolutions.

6.2.4 System correctness

The test case used for correctness results is “Gnomon in mud”. The other test cases produce similar results, and so are not included. POLYMECO is used to compare the single- and multi-CPU simulation meshes. The metric used takes the distance from each vertex on the original mesh (single CPU version) and a corresponding sampled point on the target mesh. The sampled point is the point that is approximately the closest point on the surface of the target mesh to the vertex in question(see Roy et al. [2004] for details). Using the metric, the maximum and mean Geometric difference is calculated and plotted against the number of CPUs used for the simulation in Figure 6.16. The grid resolution for the simulation is 80x120x80, which is relatively low, but POLYMECO was unable to cope with mesh sizes that were generated with higher grid resolutions. Figure 6.17 gives a plot of the Geometric deviation as a colour scale projected onto the single CPU model. The colour scale is adjusted to make regions of deviation more apparent. In general, the figure shows that there is little significant deviation. The regions with large deviation occur when the mud spatters in a different direction to the original, due to the small differences in initial conditions created by the multi-CPU implementation. The surface of the mud on the floor of the scene shows deviation, as there is marginally more fluid in the scene for the multi-CPU implementation. This is probably due to the simplification of mass distribution across slave boundaries. The negative mass left over when a cell is emptied is neglected and, thus, the overall fluid mass and volume are increased.

University of Cape Town

The maximum and mean geometric deviation, as a percentage of the bounding box diagonal plots, show that the deviation remains bounded and reasonably constant for the mean as the number of CPUs increase. Interestingly, the maximum deviation is inversely proportional to the number of CPUs. This suggests the more slave boundaries there are, the more the boundary simplifications even each other out. The maximum deviation is below 15% of the bounding box and as mentioned the main deviation occurs with droplets travelling in different paths from the initial conditions. The mean deviation is small, always below 0.6% of the bounding box diagonal.

These tests help us to verify the correctness of the simulations numerically, but do not indicate the visual satisfaction of the end product as would be seen by the visual effects artist or viewer. The animations created would need to be viewed by the artists or viewers for final verdict of qualitity and is not covered in this thesis.

6.2.5 Profiling

Using TAU, each of the simulations are profiled for different numbers of CPUs. The profiles were performed on the CHPC and interesting points on the graphs from Section 6.2.3 were profiled. Here, the caveat must be added that a profiled simulation will have slightly different behaviour to a non-profiled simulation, as the

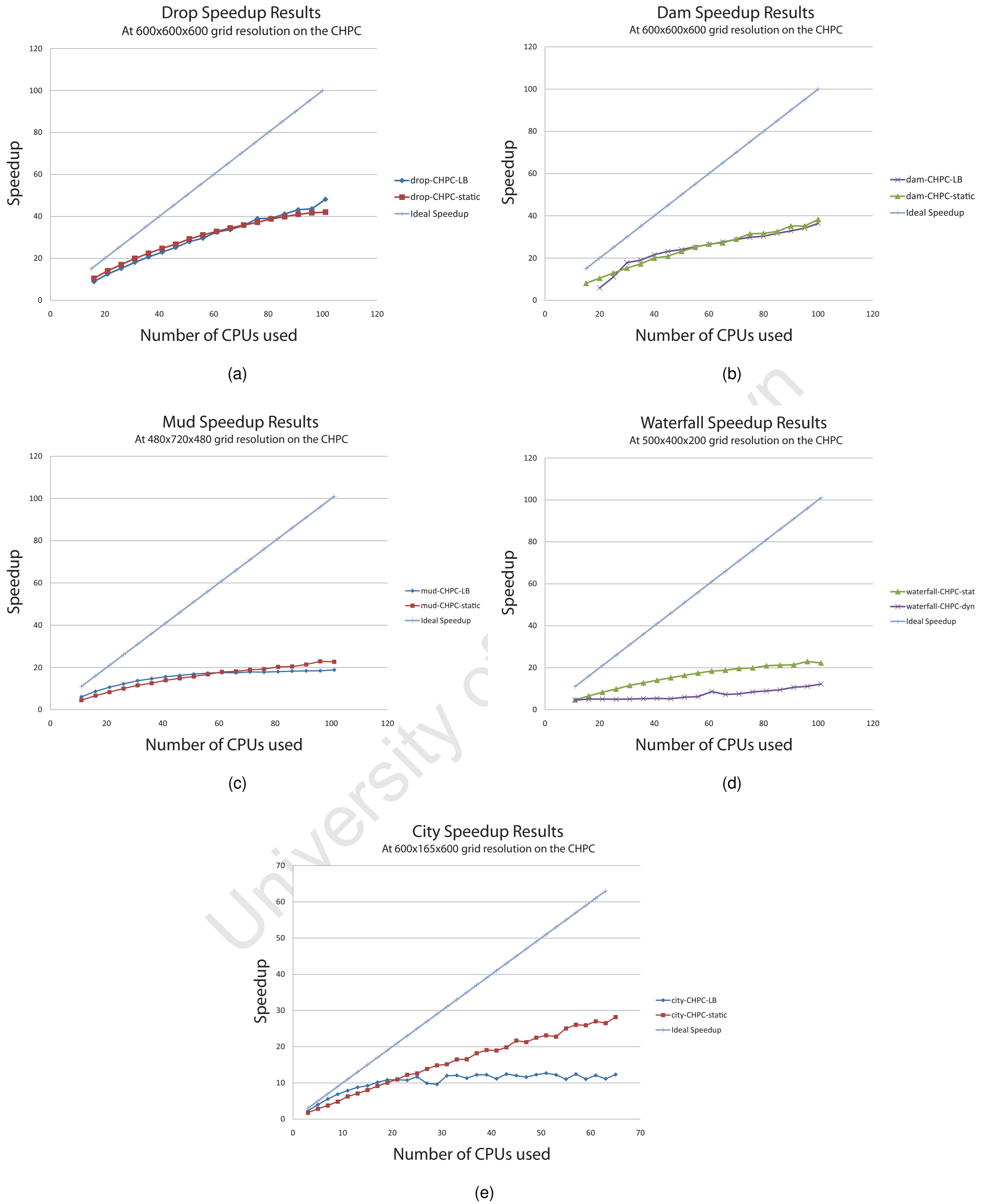


Figure 6.14: The speedup results for the tests cases at high resolutions.
The strange "kink" in the dynamic load balancing graph, for the dam simulation, is due to the initial conditions of the scene being set the same as the static load balancing. The dynamic initial conditions require larger memory than was available.

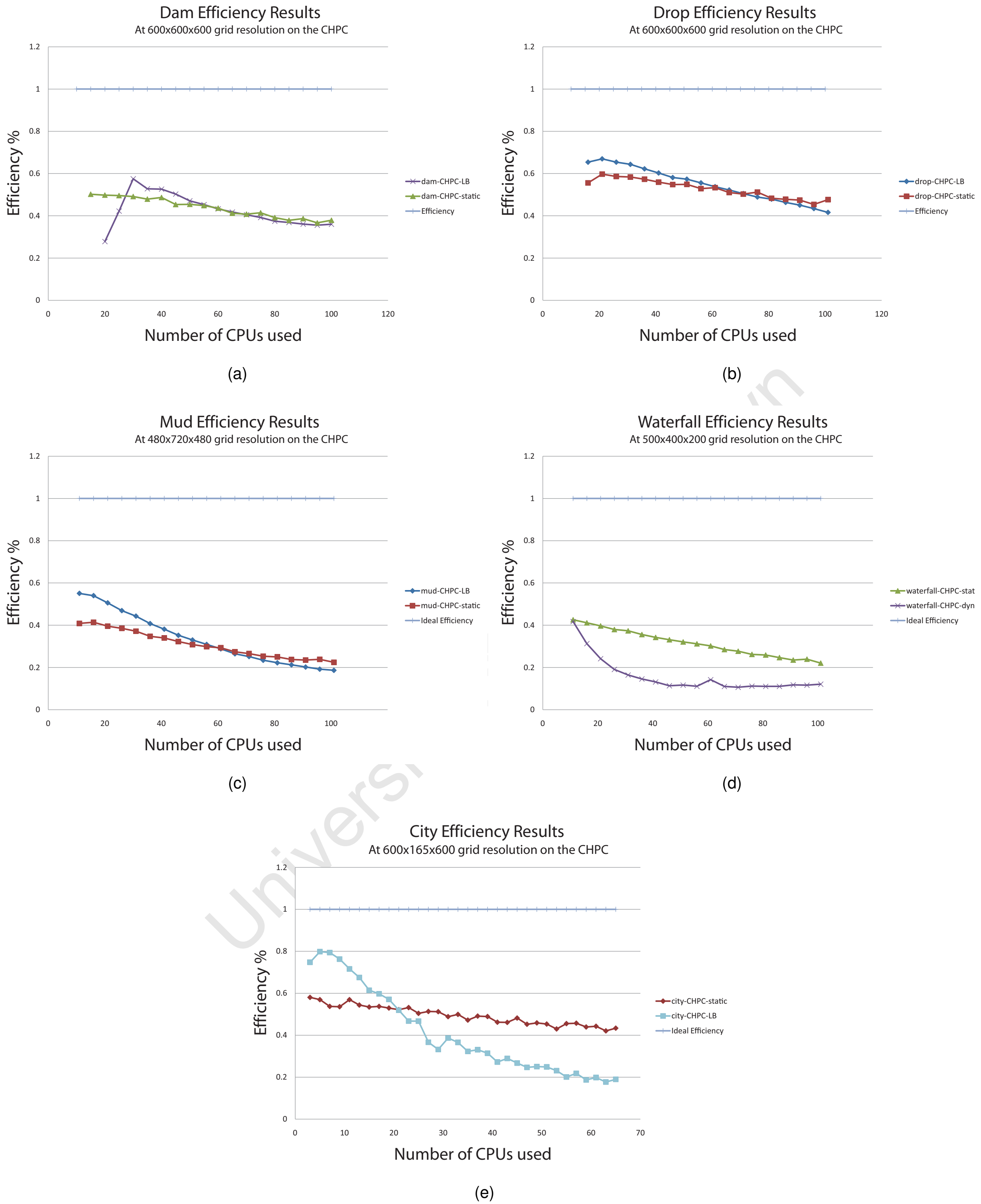


Figure 6.15: The efficiency results for test cases at high resolutions.
The strange "kink" in the dynamic load balancing graph, for the dam simulation, is due to the initial conditions of the scene being set the same as the static load balancing. The dynamic initial conditions require larger memory than was available.

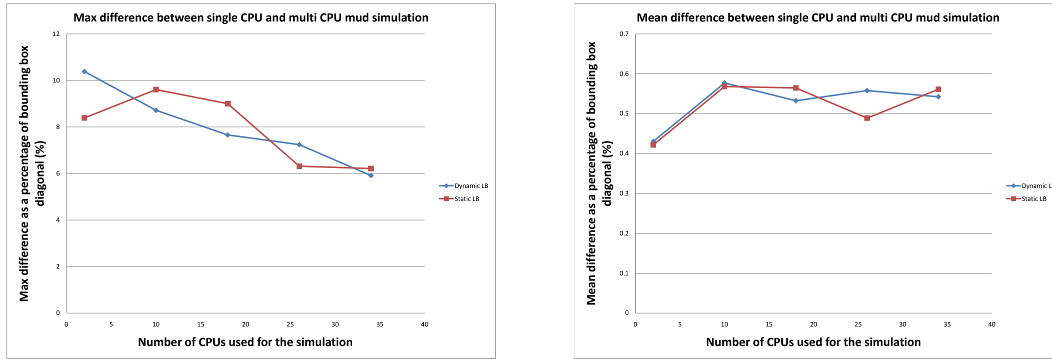


Figure 6.16: Maximum and Mean Geometric deviation for the Gnomon in mud simulation. Interestingly, the Maximum Geometric deviation decreases for higher number of CPUs. The mean deviation increases slightly from 2 to 10 CPUs and then remains reasonably constant up to 34 CPUs.

Cape Town

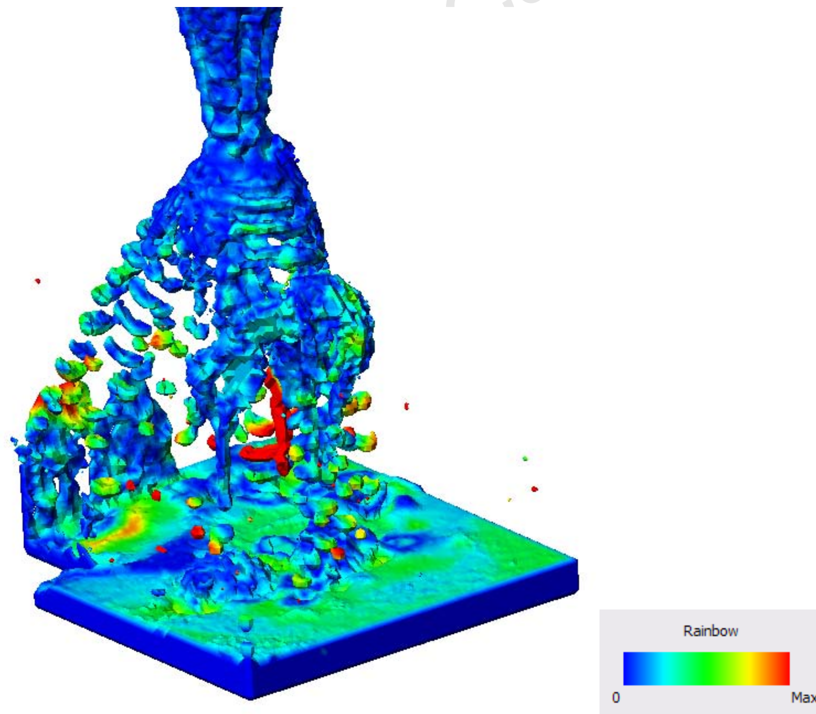


Figure 6.17: Maximum Geometric deviation for static load balancing, represented by a rainbow colour scale from blue to red projected onto the single CPU model.
The scale is modified to make the differences more apparent. The scale starts at 0, blue, and is saturated at 0.18 (or 2.7% of the bounding box diagonal). All vertices with an error above this are represented in red.

code required for profiling takes some of the running time. To minimise the differences, only key functions are profiled, and smaller functions called from the key functions are not. The simulation is broken down 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 wait times 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 is not included here, as it only accounts for 3.5% of the overall run-time of the simulation (see Section 4.5.7). For all simulations, the low resolution profiles are given in Figure 6.18. The higher resolutions exhibit similar behaviours, as seen from the previous speedup and efficiency graphs, so not all are profiled. The drop and dam simulation profiles are given in Figure 6.19. 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.

6.3 Conclusion

This chapter provides the results from the parallel implementation of the VOF LBM algorithm. Five test cases, inspired by research or movie scenes, were constructed for use during the different tests. These are the drop, dam, mud, waterfall and city test cases.

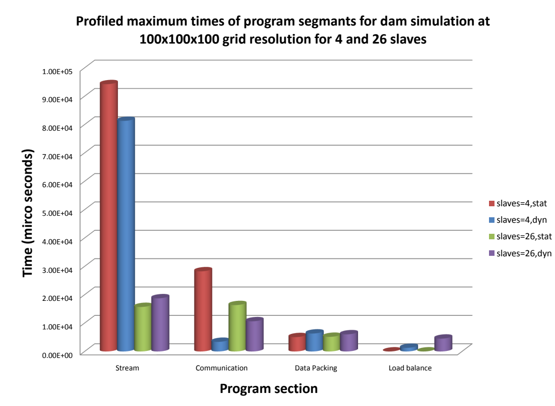
First, the rendered versions of the final simulations were shown (Section 6.2.1). These renderings made use of the Mantra ray-tracer and different combinations of textures and shaders.

Before the scalability tests for the different parallel implementations (dynamic and static load balancing) could be run, appropriate parameter values had to be chosen. Section 6.2.2 provided the graphs for simulation run-time plotted against various parameter values. From these graphs the values for the dynamic load balancing algorithm were chosen. The static load balancing does not require any parameters.

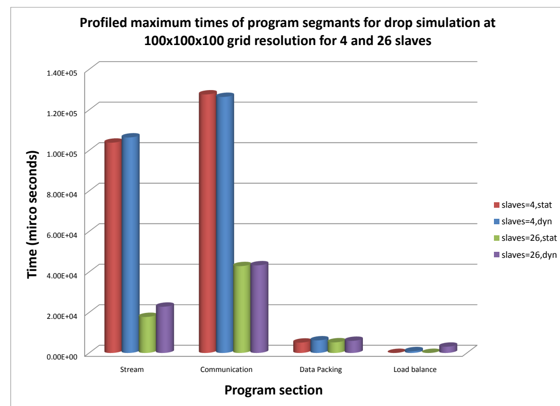
The speedup and efficiency graphs for the two implementations are provided in Section 6.2.3. These graphs show how the algorithm behaves as the number of CPUs vary. The obtained values for speedup and efficiency are given along side the ideal values for these measures to provide a means to measure the overall scalability of the algorithm.

To ensure that the parallel simulation is correct, we conducted system correctness experiments (Section 6.2.4). These compare the mesh from a single CPU VOF LBM implementation with the mesh generated by the multi-CPU implementation. These meshes should be relatively similar and contain no large differences.

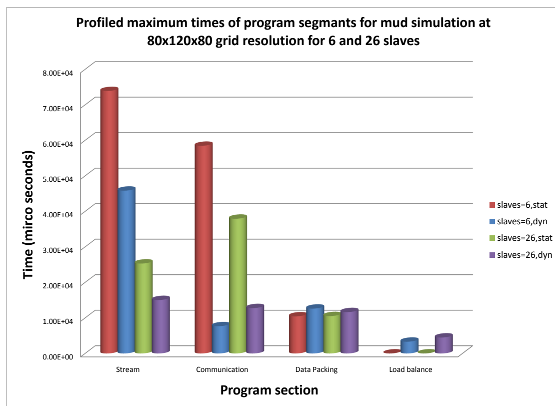
University of Cape Town



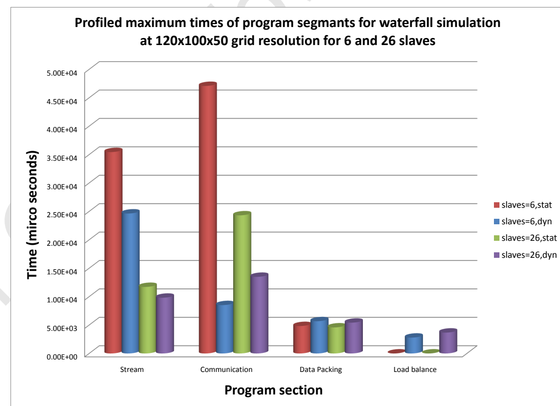
(a)



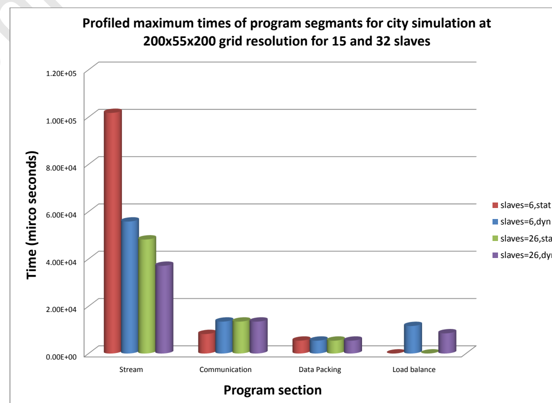
(b)



(c)

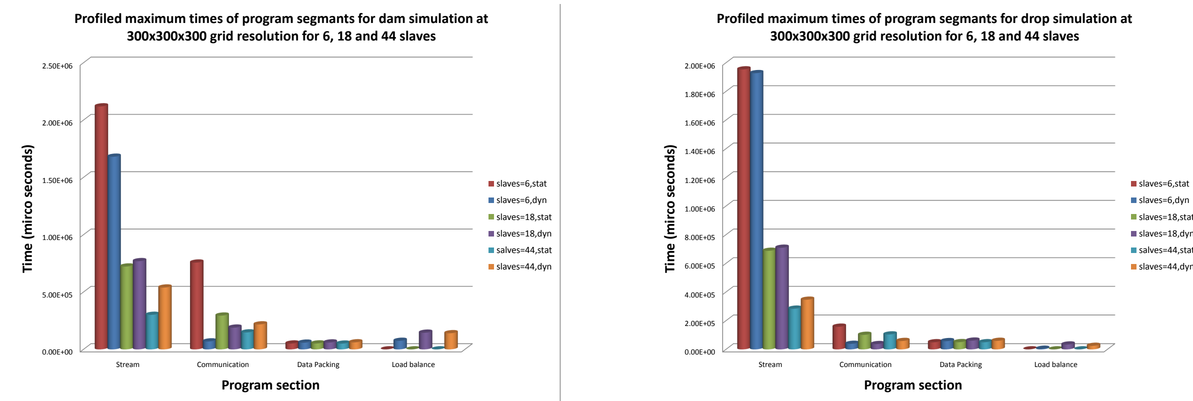


(d)



(e)

Figure 6.18: Profiling of simulations at low resolutions.
The four major sections of the simulation have been profiled and are indicated on the graphs.



(a) (b)

Figure 6.19: Profiling of drop and dam simulations at medium resolutions.
The four major sections of the simulation have been profiled and are indicated on the graphs.

University of Cape Town

Lastly, to gain insight into the reason the algorithm behaves as it does, the profiling tool TAU is used to monitor the run-time of different portions of the implementation in Section 6.2.5.

Chapter 7

Discussion

Here, we interpret the results presented in Chapter 6. We begin with comments on the general production of the final animations from start to finish (in Section 7.2), that is from scene creation through to the rendering of the final animation. The parameter selection for dynamic load balancing is discussed next in Section 7.3. Examination of the scalability and efficiency results are given in Section 7.4 and, lastly, we provide recommendations for architecture choices and feasible simulation scales.

7.1 Implementation and development

One important point to emphasize is that a parallel simulation is a very complex system, with different interacting parts and numerous algorithm synchronicities. With greater levels of complexity added by the parallel implementation and load balancing, it becomes increasingly difficult to manage "a stable release" version of the code. Here, the definition of "stable" is the same as that used in software life cycle development and differs, of course, from the stability of the algorithm from a numerical point of view. These two things are often linked. Essentially, the "stable release" version would be one which behaves as expected without any bugs. To verify this, multiple test cases are needed. For the parallel simulation, this means the full spectrum of simulations used in this thesis. These need to be run for different numbers of CPUs and parameter values. The output of the simulations then need to be verified.

This testing process has been performed for this thesis, but further testing is still warranted. The required outcome definition, namely "something that looks visually appealing", also causes problems with the testing and analysis process. For example, one bug that was found, was that the parallel version of the implementation produced more fluid for greater numbers of CPUs. This was due to an efficiency optimisation, where some cells were not informed of cell states for neighbouring cells on a slave-neighbour. The effects of this bug were not noticeable until the mud and waterfall tests cases were analysed for correctness using POLYMECO, as they appeared correct when rendered. Of course, this would skew the scalability results, so the bug needed to be fixed. This example illustrates the need for testing almost every change to the code base, to make sure that everything is running correctly.

More problems arise when a bug manifests in the parallel implementation, as a cluster environment is not

University of Cape Town

easy to debug. The changing workload division of the dynamic load balancing also complicates matters. One solution to this problem is a stricter definition of the required outcome, to more closely match physical constraints. An example of such a constraint is the conservation of mass. In this case, after each step of the algorithm (or even some sub-steps), mass could be compared against the previous step's mass. The master node in the simulation would have to collect the results from the slaves to determine that the global mass remains constant. Other approaches, such as monitoring each of the distribution functions, could be taken.

Ideally, two implementations should be created, a debug version and a release version. The debug version should contain both sanity checks and physical constraint checks, perhaps in the form of assertions. This version could then be run in an automated fashion and when a violation occurs, a violation notification could be sent.

It is important to mention here the nature of the LBM. It is a scheme that eventually arrives at fluid behaviour consistent with the Navier-Stokes equations. It is not a direct numerical solution to these equations. The proof of equivalence makes use of a number of assumptions (Appendix B). The correctness of these assumptions can be strongly argued, but nevertheless the scheme is somewhat decoupled from the exact Navier-Stokes equations. The origin of the LBM was also in part based on trial and error (Section 2.4.1.2). Researchers found what worked and used that. Indeed, the stability analysis that has been researched (Section 3.2.3) is not complete and the algorithm is not fully understood. This makes it harder to be exactly sure what physical interactions should be taking place and, thus, harder to ensure physical correctness in the face of inadvertant bugs.

The same statement can be made for the VOF method of Thurey et al. [2006]. It is not fully defined and the relation to physical effects are "rule of thumb" methods rather then direct physical equations. Nonetheless this surface method lends itself well to the LBM and because of its simplicity, constraint checking can be added easily.

A strong advantage of the LBM method, is that all effects are localised to a certain grid region. For example, if an incorrect value were to occur in one cell, causing corruption in the physical properties, then it would take time to propagate to other cells. It is easy to locate such occurrences and the exact error. The simple distribution interactions allow the discovery of the root cause far more easily. Other methods (e.g. [Stam, 1999]) require global pressure correction and other complicated numerical schemes, which make it more difficult to track down bugs.

University of Cape Town

7.2 Animation Creation

There are four steps to the creation of the animation: the scene specification, animation export, simulation on the cluster and rendering. The first step was seamless and straightforward in all cases. The second step yielded a performance problem, which will be discussed shortly. The third is discussed later in section 7.4 and the last was also problematic. It was due to the export and rendering steps that attempts to use higher grid sizes did not succeed. The drop and dam simulations had the largest number of cells, with a grid size of $600 \times 600 \times 600$.

The animation export requires calculating the intersection of each grid cell with the objects in the scene.

This is a $O(n^3)$ algorithm, assuming a constant cuboid domain with side length n. Each operation uses the ray casting function provided by the Houdini Toolkit. Ray casting is a expensive operation in itself, so the export is lengthy. For the complex geometry case of the waterfall, this took over 27 hours. This procedure could be improved by scanning individual axis aligned lines, rather then individual cells. Currently, a ray is fired in an arbitrary direction for each cell in the grid and if it intersects the object an even number of times then the cell is outside the object. If it intersects an odd number of times, then the cell is inside the object. The object is assumed to be suitably convex. Instead, a ray could be fired for each cell in the xy-plane along the z-axis. This would result in k points of intersection with the object. Firstly, the grid cells between intersection 1 and 2 would be marked as inside the object. The algorithm is continued for each pair of intersection points, until $k-1$ to k has been reached. This would reduce the algorithm to $O(n^2)$. It may also be possible to borrow other ray optimisations from ray tracing.

The final rendering of the fluid mesh also presents a problem, as the renderer can only handle meshes up-to 600MB in Houdini's format, at least on a computer with 3GB memory. For the mud scene, the meshes generated per frame, in condensed format (the format output from the cluster back-end), are approximately 200MB per frame. This then needs to be converted into the Houdini format, which triples the mesh size to 600MB per frame. Naturally, at the beginning of the scene there was less fluid and the mesh size became larger for each subsequent frame. Additionally, the type of shader used for the mesh plays a role in the rendering process. For example, when using a lava shader, mantra quickly consumed over 3GB of memory and failed for an input mesh size of 100MB, but when using a simpler water shader, it rendered easily.

The first step in resolving the mesh size problems involved using a special delayed mesh loader. This caused mantra, the renderer, to load the mesh only when ray-tracing required the mesh. Although this helped with stability during rendering, it did not solve the problem. The real mechanism required is a custom delayed mesh loader, which could use an optimized structure such as a Quad-tree. The leaves of the quad tree could be a section of the scene, a cube of size $10 \times 10 \times 10$ grid cells for example, which are loaded on demand. This would be limited by the speed of the hard disk accesses, but this is unavoidable. Another possibility is to completely distribute the rendering of the scene. The slave processes could then calculate ray intersections for their section of the scene. However, such a very tightly coupled simulation and rendering architecture would be very complex.

We recommend using mesh compression or simplification, coupled with an optimized structure and a delayed loader. This would be sufficient for far larger resolutions, without causing too many additional code complexities. The preparation of the compressed mesh and the optimised structure could be partially performed by each of the current slave processes. Thus, the computation remains partially distributed. Of course, some final assembly of the optimized structures from each of the slaves would be required.

Lastly, instability problems became an issue at large spatial resolutions for the drop test case. This occurs because the drop has more distance to cover while being accelerated by gravity. The overall fluid speed then becomes too high for the VOF LBM to simulate in a perfectly physically correct way. This manifests itself as an "explosion" of the drop before it hits the water. Thurey et al. [2005a] use time scaling when velocities become to large. For the multi-CPU version, this requires time scaling across all slaves. The approach taken by d'Humières et al. [2002] modifies the relaxation step, a local operation. This approach would more easily fit in the current system design, as no extra slave communication would be necessary.

University of Cape Town

7.3 Dynamic Load balancing parameters

The load balancing parameters provide a means to fine tune appropriate times of load balancing. The optimal values will differ for each simulation. The computation threshold has been shown, in Section 6.2.2, to have the greatest influence on the speed of the simulations, followed by the LB interval and lastly the wait factor. The results show no consistent minimum and the final choice of parameter values is a balance between making the load balancing decisions sensitive to an imbalance (good for simulations such as the dam) or insensitive (good for simulations such as the drop). As a design decision, we choose an LB interval of 1, favouring unbalanced simulations. The wait factor is also chosen as 1, but as stated does not play a large role in the simulations.

For the CRUNCH architecture, which has a Gigabit Ethernet interconnect, the computation threshold should be set between $0.013 - 0.225$. To favour both unbalanced and balanced simulations a higher value should be used. On the CHPC architecture, the range is $0.03 - 0.11$. The range is far smaller, as the interconnect is faster Infiniband. Both architectures benefit from sensitive load balancing. As we wish to investigate the effects of load balancing, we choose to use a value of 0.035 for both architectures. This will make the run times poorer in some situations, but it improves our ability to analyse the effects of load balancing. Ideally, it would be best to have some mechanism to determine the most appropriate value based on the initial conditions or possibly have dynamic values that change during the course of the simulation. The parameters could be functions of the fluid cell count for each slave. When an imbalanced count is detected, sensitive parameter values could be selected and the opposite for balanced counts

University of Cape Town

7.4 Scalability and Efficiency

7.4.1 Test case discussions

For this section of the discussion, the simulations are grouped, as some exhibit similar characteristics. The drop and dam simulations are discussed together and the mud and waterfall are grouped, but the city simulation is presented singularly.

For the drop and dam simulations, even though the starting conditions are very different, the drop with a reasonably balanced load and the dam with a relatively unbalanced load, the scalability is rather similar. As expected, the dam simulation benefits more from dynamic load balancing than the drop simulation. However, the benefit is only with a lower number of CPUs and there is a crossover point where the static load balancing scales better. At low resolutions the scaling is not particularly good, while for the medium and high resolutions the scaling is far better.

Profiling (Figure 6.19) shows the reasons for these differences. Firstly, Sub-Figure (a) shows that the static load balancing has an unbalanced load for low numbers of CPUs and becomes far better for higher values. 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. The sub-figure shows that ultimately the balance becomes poorer for large numbers of CPUs. This is most probably due to the fact that slaves only know about the time they spend waiting for neighbours and the staggered computation times across the whole domain prevent an unbalanced slave on the one end of

the domain from claiming work from a slave on the other end of the domain. For example consider a simulation with 10 slaves. Let slave 2 have computation time of x secs 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 $0.66x$, which is a definite candidate for load balancing, but the difference between its neighbours is only 5% in terms of computation time, so it will not ask to load balance.

In the end, the Figure 6.19 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 effectively non-parallelizable time (constant for all slaves). The other factor causing poor scalability of the dynamic simulation is the time required to load balance, which increases with CPU number. 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 wait times 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.

Sub-Figure 6.19 (b) shows the same graph, but for the drop simulation. It exhibits very similar characteristics, except the balance in general is far better. 6.18 shows the same results at a lower resolution. Similar effects are observed, 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 Figure 6.18 (c) and (d) 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 Figure 6.18 (e) demonstrates that the static case is very unbalanced, while the dynamic case has a better balance. This is understandable when the scene specification is considered (Section 6.1). The scene is specified with some 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, the dynamic load balancing algorithm performs better for this test case, as it measures such an imbalance and adjusts slave-domains accordingly.

University of Cape Town

7.4.2 General Simulation

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. For the lower CPU number range, 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 and, thus, the entire simulation is slowed down. As mentioned, this is because only local slave computation times are considered in balancing. Incorporating a higher order scheme that regulates the load balance with global knowledge would be highly 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 [Dachsel et al., 2007]. Conversely, strong scaling fixes the 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.

It is important to note that the VOF LBM algorithm requires a large amount of communication every iteration (twice in fact, see Section 5.3.2). Therefore, good scaling is not expected. With this in mind, the scaling results for our implementation are positive.

7.5 Architecture and scene scale recommendations

The main factor that causes poor scalability is the communication time. This causes very poor scalability on the CRUNCH architecture. The network architectures for CHPC and CRUNCH are very different, with CRUNCH using Gigabit Ethernet and CHPC using Infiniband. Table 5.1 shows that the latency time for Infiniband is 18.55 times faster than Gigabit Ethernet. Bandwidth is 3.33 times larger with Infiniband. This is a great advantage for the VOF LBM, as the performance is strongly limited by the communication. In fact, it does not make sense to use a large number of CPUs for simulations without such an interconnect. This is due to poor efficiency in synchronisation.

The size of animations that are being created in current research and, hence, movies is similar to that of the high resolution tests (see resolutions in [Losasso et al., 2008]). The scaling at these resolutions is used to make recommendations about the amount 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.

From the wall clock times in Section D.2, 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. However, this is

University of Cape Town

still not ideal and faster times would be better. With 400 CPUs, this figure becomes 2.83hrs. The returns are diminishing for the number of CPUs. For this number of CPUs, further optimisations of communication are required, as that forms the main bottleneck.

Chapter 8

Conclusions

8.1 Thesis Aims

Here, the aims from Chapter 1 are mentioned along with a brief summary of the extent to which these aims were achieved.

1. *To evaluate the robustness of the VOF LBM for simulating fluids with a free surface*. The VOF LBM algorithm proved not to be as robust as we hoped, but fluid behaviour is adequately modelled by the algorithm. The stability of the algorithm is sensitive to the input parameters and the surface becomes unrealistically bumpy for a large subset of the parameters.
2. *To estimate the simulation scales and spatial resolution that can be created using a single CPU implementation of the VOF LBM;* Due to stability constraints, once the viscosity of the fluid is chosen, the rest of the parameters, such as the space and time steps, are relatively fixed (a small range for these parameters maintains algorithm stability). As such, fluids have to be simulated at a small spatial resolution. To simulate water in a stable way, the spatial distance from the center of one cell to the center of a neighbour cell must be in the order of $0.0004m$. The realistic simulation scale that can be simulated on a single CPU is in the range 200-300 grid cells. Combining these limits, the area of water that can be realistically simulated is between 8-$12cm$. To simulate a city being flooded, these small distances would need to be super imposed on a large city. In addition, the memory requirements also make it infeasible to use a single PC for simulations.
3. *To produce large scale simulations of fluid;* The combination of Houdini with a plug-in that was loosely coupled to a cluster simulation produced the five different animations shown in Section 6.2.1. These were all produced at a high grid resolution, the lowest being the city animation at a resolution of $600 \times 165 \times 600$. While rendering of large cities does present problems, we did effectively render scenes of this size. At these larger resolutions, the volume of fluid showed artifacts along the sides of the walls, namely, unphysical striping. This could be due to instability of the algorithm at the boundaries or faults in the mass distribution of the VOF method. Algorithm instabilities were apparent when large velocities were present, notably in the drop test case when the drop "exploded" before hitting the water.
4. *To compare the performance and scalability, with respect to the number of CPUs, of the static and dynamic load balancing implementations of the VOF LBM;* At low resolutions both the dynamic and static

University of Cape Town

load balancing algorithms showed poor scaling. For the unbalanced cases, dynamic load balancing was able to increase the efficiency of the simulations. However, the increase in performance was not dramatic. In addition, for a high number of CPUs, dynamic load balancing performed badly, since it only considers local information and the maximum run-time of a single slave was higher than that of static load balancing. For medium and high resolutions, the algorithms scale better. The differences in the load balancing algorithms were more prevalent at these resolutions, but again for large numbers of CPUs the static load balancing out performed the dynamic.

The profiling in Section 6.2.5 shows that the communication time is the main cause of the poor scaling at higher numbers of CPUs. Essentially, this time contributes to the non-parallel portion of the code. Hence, Amdahl's law, which states that the scalability is limited in cases where the sequential portion of code for an algorithm is high, explains this poor scalability. However, with the amount of communication required, the scaling results for the implementations are positive.

5. *To recommend required architecture and simulation scales which can be created using a multi-CPU version of the VOF LBM*. Due to the communication time, a faster interconnect between the slaves is recommended. The Infiniband interconnect showed vast improvements over the Gigabit Ethernet. If only a small number of CPUs are being utilised, then Gigabit Ethernet will be also be adequate, as the ratio of computation to communication will be high and the interconnect will play a smaller role. Due to the high memory utilisation of the method, architectures with large amounts of cache are recommended.

   To produce large scale simulations (approximately $600 \times 600 \times 600$ in grid size and 1 minute in length) in a reasonable time of under 5 hours, it is estimated that a cluster with 200 CPUs running at 2.6Ghz would be required. More CPUs would yield shorter times, but diminishing performance per CPU must be expected.

University of Cape Town

8.2 Recommendations and Future work

1. *Improve the simulation framework*. The first recommendation we make is a self checking simulation framework. This is very important when future optimisations and new features are added to keep the simulation code in a stable form. This could be in the form of debug code that monitors the physical constraints and asserts when they are violated.

2. *Improve the algorithm stability*. The stability of the LBM algorithm can be improved. There are two recommended approaches by d'Humières et al. [2002] and Thurey et al. [2005a]. We favour the former method, as it modifies local distribution functions as they relax to equilibrium. This is a suitable extension to the current algorithm, without modification and added communication. The second requires time scaling of the distribution functions, a global operation that requires all slaves to be scaled.

3. *Algorithm adaptions due to hardware considerations*. Cache optimisations and special hardware or instructions could add significantly to performance. The current implementation must load the entire LBM structure into memory to do some basic checks, such as whether a cell is empty. This means that all the distribution functions would be loaded into cache as well as the cell flag type, even though the distribution functions are unnecessary. This can be optimised by only loading the cell flags to do the check. However, this would increase code complexity. Wilke et al. [2003] optimise LBM code with respect to different levels of cache by operating on groups of cells that all fit into cache at one time. Streaming SIMD

Extension (SSE) instructions are available on current CPU's and can do multiple floating point operations per instruction [Ramanathan et al., 2006]. Four floating point operations can be performed at once, with a theoretical fourfold speedup. However, additional complexities in data handling are involved. Lastly, the use of hardware such as a GPU can be applied to the simulation.

4. *Better mesh and LBM grid management to cope with large scenes.* The largest animations in this thesis utilized a grid size of $600 \times 600 \times 600$. Larger gird sizes yielded output fluid meshes that the Mantra renderer was unable to handle. Mesh compression and a delayed mesh structure would allow the renderer to query parts of the scene at render time. Combining the distributed simulations with a distributed renderer would be ideal. Structures, such as quad trees, could be incorporated to represent the LBM grid structure more effectively. Dynamic memory management would also be an advantage.

5. *Global information usage for load balancing.* The dynamic load balancing algorithm could be adapted to make use of more global information. Currently, the dynamic load balancing implementation has large maximum individual slave run-times when utilising large numbers of CPUs. This can be alleviated by incorporating the computation times of all the slaves into the load balancing decision process.

6. *Dynamic objects should be introduced into the simulation.* Objects that accelerate and can be accelerated by the fluid are interesting. However, the distributed implementation of such a system would be complex, as the object's movement would need to be tracked across all slave boundaries and feedback forces would need to be taken into account.

Though a number of improvements can be made to the system presented here, as detailed above, it can be used as is for simulations of liquids at high resolutions with Houdini.

University of Cape Town

Appendix A

Advanced Fluid Effects

So far we have talked about the different numerical methods to model the fluid dynamics and approaches to representing fluids with free surfaces. One could use each of the methods to produce a tool for animators to add fluid to a 3D scene. However, there is much research into high level effects using fluid dynamics that are very advanced physical systems or beyond pure physical simulation and give the animator further control of the fluid behaviour. Here, we give a brief summary of these advancements and to which of the three techniques they have been applied.

Table A.1 gives an overview of the effects. This list in by no means exhaustive, but gives insight into the current advanced research being done in the fields. **Two-Way Solid fluid coupling** is important as fluids apply pressure and have pressure applied by objects. The objects do not necessarily align with grid spaces or exist at the sub grid level. **Mesh Driven Control** provides animators with a mechanism to use a mesh as a control to contain fluid or as a target for moving fluid. These methods aim to retain small scale fluid like behaviour while giving animators high level control. **Multi-resolution control** describes at which level of detail an animator induce control at multiple resolutions.

Fluids do not only interact with solid objects and **Fluid-Fluid interaction** is important. These simulations are able to have two types of fluid, such as water and oil interacting in physically correct ways. **Multi-phase fluids** is the next step, where the fluids are able to change from one phase to another, e.g. ice melting to water or water changing to steam. **Two-way Deformable object fluid coupling** incorporates methods where deformable objects can be fully integrated into the fluid simulation.

An extra level of realism can be added with **Bubbles**. Air within the fluid is properly simulated. In some cases, real time simulation speeds are required. **Interactive** simulations provide this functionality. Sometimes **Added Turbulence control** is needed. There are cases where an animator would like to add more turbulence to fluid, such as the wake behind a motor boat. Extra special effects like melting wax and fire are considered in the **Melting and burning** section. The **Fine** spray section is for effects such as mist, which can add a much needed level of realism.

University of Cape Town

	Technique		
	Eulerian		Lagrangian
Advancement	Direct NS	Indirect NS	
Two-Way Solid fluid coupling	Carlson et al. [2004]	Thurey et al. [2006]	Premoze et al. [2003], Keiser et al. [2005, 2006]
Mesh Driven Control	Shi and Yu [2005], Fattal and Lischinski [2004]	Thürey et al. [2006]	Thürey et al. [2006]
Multi-resolution Control		Thürey et al. [2006]	
Fluid-Fluid Interaction	Losasso et al. [2006b]		Müller et al. [2005], Mao and Yang [2006]
Multi-phase fluids	Nguyen et al. [2002], Losasso et al. [2006b]		
Non-regular Grid For complex Geometry	Chentanez et al. [2007]		Implicit
Two-way Deformable object fluid coupling	Guendelman et al. [2005]		
Bubbles	bubbles alive	Thürey [2003]	Müller et al. [2005]
Interactive	Stam [1999]	Thurey et al. [2005b]	Müller et al. [2003]
Added Turbulence control	Selle et al. [2005]		
Melting and burning	Carlson et al. [2002], Losasso et al. [2006a], Nguyen et al. [2002]		
Fine spray	Losasso et al. [2008]		

Table A.1: An overview of the advanced fluid methods that have been researched by different authors.

University of Cape Town

Appendix B

Proof of physical correctness

Here, it will be shown that the discrete 2D lattice mentioned above with the defined collision operator and discrete equilibrium distribution will produce the differential equation 2.2 in the velocity, u, and pressure, P. The proof is shown for the two dimensional case, but is analogous in three dimensions. The structure of the proof follows the outline in Chen and Doolen [1998].

The basis of the proof relies on the Chapman-Enskog or multiscale expansion of the LBE (equation 3.18). The expansion first uses a 2D second order Taylor expansion to approximate $f_i(\overrightarrow{x} + c_i\Delta t, t + \Delta t)$ around $(\overrightarrow{x}, t)$. Then, the f_i are split up into the sum of the equilibrium function and perturbations on smaller and smaller scales. This split induces new definitions on time and space scales. Each scale, in some sense, represents changes at different magnitudes with respect to the distribution functions. Once we have a new expression of the LBE expanded to take into account effects on different scales, the assumption is made that the scales interact independently and each scale can be separated out to perform further analysis. Having derived new approximate equations at different scales, the velocity and pressure are obtained from equations 3.30 and 3.31. The different scales are then combined to discover the appropriate differential equations.

By applying the Taylor expansion to the LBE equation we get

$$\begin{aligned}\frac{f_i^e - f_i(\overrightarrow{x}, t)}{\tau} \quad = \quad & \frac{\partial f_i(\overrightarrow{x}, t)}{\partial t}\Delta t + \frac{\partial f_i(\overrightarrow{x}, t)}{\partial \overrightarrow{x}}(\overrightarrow{c_i}\Delta t) + \\ & \frac{1}{2}\left[\frac{\partial^2 f_i(\overrightarrow{x}, t)}{\partial t^2}(\Delta t)^2 + 2\frac{\partial f_i(\overrightarrow{x}, t)}{\partial t \partial \overrightarrow{x}}(\overrightarrow{c_i}\Delta t)(\Delta t) + \frac{\partial^2 f_i(\overrightarrow{x}, t)}{\partial \overrightarrow{x}^2}(\overrightarrow{c_i}\Delta t)^2\right] + \\ & O(\partial^3 f_i(\overrightarrow{x}, t))\end{aligned} \quad \text{(B.1)}$$

For simplicity, let $\Delta t = 1$, drop the $O(\partial^3 f_i(\overrightarrow{x}, t))$ terms and introduce the following notation: $f_i = f_i(\overrightarrow{x}, t)$ and $\frac{\partial}{\partial \overrightarrow{x}} = \nabla$. With these mentioned simplifications, Equation B.1 becomes

$$\frac{f_i^e - f_i}{\tau} = \frac{\partial f_i}{\partial t} + \nabla f_i \overrightarrow{c_i} + \frac{1}{2}\left[\frac{\partial^2 f_i}{\partial t^2} + 2\nabla \frac{\partial f_i}{\partial t}\overrightarrow{c_i} + \nabla^2 f_i \overrightarrow{c_i}\,\overrightarrow{c_i}\right] \quad \text{(B.2)}$$

Now, let

$$f_i = f_i^e + \epsilon f_i^1 + \epsilon^2 f_i^2 + O(\epsilon^3) \quad \text{(B.3)}$$

University of Cape Town

where $f_i^k \ll f_i^{k-1}$. The value ϵ can be viewed either as a very small value or a marker that can be used to indicate which scale a given term belongs to [Wolf-Gladrow, 2000]. In the former approach, ϵf_i^1 is seen as very small, while in the latter f_i^1 is very small and this is marked by ϵ. For the proof we will only consider terms up to and including ϵ^2. The following constraints must hold to conserve physical quantities:

$$\sum_i f_i^e = \rho, \qquad \sum_i f_i^e \overrightarrow{c_i} = \rho \overrightarrow{u} \tag{B.4}$$

and for $k > 1$

$$\sum_i f_i^k = 0, \qquad \sum_i f_i^k \overrightarrow{c_i} = 0. \tag{B.5}$$

For the quantities to be properly conserved at each scale, the following substitution for the time and space is used:

$$t = \epsilon t_1 + \epsilon t_2, \qquad x = \epsilon x_1. \tag{B.6}$$

The partial derivatives then become [Wolf-Gladrow, 2000]

$$\frac{\partial}{\partial t} = \epsilon \frac{\partial}{\partial t_1} + \epsilon^2 \frac{\partial}{\partial t_2}, \qquad \frac{\partial}{\partial x} = \epsilon x_1. \tag{B.7}$$

Substituting B.7 into B.2 we get

$$\epsilon \frac{\partial f_i}{\partial t_1} + \epsilon^2 \frac{\partial f_i}{\partial t_2} + \epsilon \nabla_1 f_i \overrightarrow{c_i} + \frac{1}{2}\epsilon^2 \frac{\partial^2 f_i}{\partial t_1^2} + \frac{1}{2}\epsilon^4 \frac{\partial^2 f_i}{\partial t_2^2} + \epsilon^2 \nabla_1 \frac{\partial f_i}{\partial t_1} \overrightarrow{c_i} + \epsilon^3 \nabla_1 \frac{\partial f_i}{\partial t_2} \overrightarrow{c_i} + \frac{1}{2}\epsilon^2 \nabla_1^2 f_i \overrightarrow{c_i}\overrightarrow{c_i} = \frac{f_i^e - f_i}{\tau}, \tag{B.8}$$

with $\nabla_1 = \frac{\partial}{\partial \overrightarrow{x_1}}$. Dropping terms with ϵ^k, $k > 2$ it becomes

$$\epsilon \frac{\partial f_i}{\partial t_1} + \epsilon^2 \frac{\partial f_i}{\partial t_2} + \epsilon \nabla_1 f_i \overrightarrow{c_i} + \frac{1}{2}\epsilon^2 \frac{\partial^2 f_i}{\partial t_1^2} + \epsilon^2 \nabla_1 \frac{\partial f_i}{\partial t_1} \overrightarrow{c_i} + \frac{1}{2}\epsilon^2 \nabla_1^2 f_i \overrightarrow{c_i}\overrightarrow{c_i} = \frac{f_i^e - f_i}{\tau}. \tag{B.9}$$

Applying B.3 up to order 2, this becomes

$$\epsilon \frac{\partial f_i^e}{\partial t_1} + \epsilon^2 \frac{\partial f_i^1}{\partial t_1} + \epsilon^2 \frac{\partial f_i^e}{\partial t_2} + \epsilon \nabla_1 f_i^e \overrightarrow{c_i} + \epsilon^2 \nabla_1 f_i^1 \overrightarrow{c_i} + \frac{1}{2}\epsilon^2 \frac{\partial^2 f_i^e}{\partial t_1^2} + \epsilon^2 \nabla_1 \frac{\partial f_i^e}{\partial t_1} \overrightarrow{c_i} + \frac{1}{2}\epsilon^2 \nabla_1^2 f_i^e \overrightarrow{c_i}\overrightarrow{c_i} = \frac{-\epsilon f_i^1 - \epsilon^2 f_i^2}{\tau}. \tag{B.10}$$

Note here that the right hand side, the collision term, is simplified to $\frac{-f_i^1 - f_i^2}{\tau}$, which agrees with the invariance of the collision operator when the DFs are at equilibrium. Consider the terms of order ϵ. This produces

$$\frac{\partial f_i^e}{\partial t_1} + \nabla_1 f_i^e \overrightarrow{c_i} = \frac{-f_i^1}{\tau}. \tag{B.11}$$

$$\frac{\partial^2 f_i^e}{\partial t_1^2} + \nabla_1 \frac{\partial f_i^e}{\partial t_1} \overrightarrow{c_i} = \frac{-1}{\tau} \frac{\partial f_1^e}{\partial t_1}. \tag{B.12}$$

To get the first of the continuity equations Equation B.11 is summed over all i and the result is

$$\frac{\partial \rho}{\partial t_1} + \nabla_1 \rho \overrightarrow{u} = 0. \tag{B.13}$$

University of Cape Town

The right hand side vanishes by B.5. Under the assumption of constant density the zero divergence rule is obtained:

$$\nabla \cdot u = 0 \tag{B.14}$$

Now, after multiplying by $c_i$and taking the sum of all i

$$\frac{\partial \sum_i c_i f_i^e}{\partial t_1} + \sum_i c_i \nabla_1 f_i^e \overrightarrow{c_i} = \frac{-\sum_i c_i f_i^1}{\tau} \tag{B.15}$$

$$\frac{\partial \rho u}{\partial t_1} + \sum_i \begin{pmatrix} c_{ix}c_{ix}\frac{\partial f_i^e}{\partial x} + c_{ix}c_{iy}\frac{\partial f_i^e}{\partial y} \\ c_{iy}c_{ix}\frac{\partial f_i^e}{\partial x} + c_{iy}c_{iy}\frac{\partial f_i^e}{\partial y} \end{pmatrix} = 0 \tag{B.16}$$

$$\frac{\partial \rho u}{\partial t_1} + \sum_i \begin{pmatrix} \nabla \begin{pmatrix} c_{ix}c_{ix}f_i^e \\ c_{ix}c_{iy}f_i^e \end{pmatrix} \\ \nabla \begin{pmatrix} c_{iy}c_{ix}f_i^e \\ c_{iy}c_{iy}f_i^e \end{pmatrix} \end{pmatrix} = 0 \tag{B.17}$$

$$\frac{\partial \rho u}{\partial t_1} + \begin{pmatrix} \nabla \begin{pmatrix} \sum_i c_{ix}c_{ix}f_i^e \\ \sum_i c_{ix}c_{iy}f_i^e \end{pmatrix} \\ \nabla \begin{pmatrix} \sum_i c_{iy}c_{ix}f_i^e \\ \sum_i c_{iy}c_{iy}f_i^e \end{pmatrix} \end{pmatrix} = 0 \tag{B.18}$$

Using the equilibrium distribution in Equation 3.19 with $c = 1$ we get the following results:

$$\sum_i c_{ix}c_{ix}f_i^e = \frac{\rho}{3} + u^2\rho \tag{B.19}$$

$$\sum_i c_{ix}c_{iy}f_i^e = \rho u_x u_y \tag{B.20}$$

$$\sum_i c_{iy}c_{iy}f_i^e = \frac{\rho}{3} + u^2\rho \tag{B.21}$$

Substituting into Equation B.18, then

$$\frac{\partial \rho u}{\partial t_1} + \begin{pmatrix} \nabla \begin{pmatrix} \frac{\rho}{3} + u_x^2\rho \\ \rho u_x u_y \end{pmatrix} \\ \nabla \begin{pmatrix} \rho u_x u_y \\ \frac{\rho}{3} + u_y^2\rho \end{pmatrix} \end{pmatrix} = 0 \tag{B.22}$$

Using Boyle's law, the pressure, P, is $\frac{RT}{m}\rho = c_s^2\rho = \frac{1}{3}\rho$ and assuming that ρ is constant, which is the case for incompressible fluids, for the x co-ordinate of the above equation we have

$$\rho\frac{\partial u_x}{\partial t_1} + \frac{\partial P}{\partial x} + \rho u_x \frac{\partial u_x}{\partial x} + \rho u_y \frac{\partial u_x}{\partial y} = 0 \tag{B.23}$$

Similar results will be obtained for the y co-ordinate. The resulting equation is known as Euler's equation for incompressible fluids and is correct to first order in ϵ,

$$\frac{\partial \overrightarrow{u}}{\partial t} + \frac{1}{\rho}\frac{\partial P}{\partial x} + \overrightarrow{u}\nabla\overrightarrow{u} = 0. \tag{B.24}$$

As $\overrightarrow{u}$ is a vector field, ∇ is the jacobian as defined in Chapter . To get the diffusion term with viscosity, the second order terms in ϵ of Equation B.10 need to to be considered, namely

University of Cape Town

$$\frac{\partial f_i^1}{\partial t_1} + \frac{\partial f_i^e}{\partial t_2} + \nabla_1 f_i^1 \vec{c_i} + \frac{1}{2}\frac{\partial^2 f_i^e}{\partial t_1^2} + \nabla_1 \frac{\partial f_i^e}{\partial t_1}\vec{c_i} + \frac{1}{2}\nabla_1^2 f_i^e \vec{c_i}\vec{c_i} = \frac{-f_i^2}{\tau}. \tag{B.25}$$

Using Equations B.11 and B.12 the simplification,

$$\frac{\partial f_i^e}{\partial t_2} + (1 - \frac{1}{2\tau})\left(\vec{c_i}\nabla f_i^1 + \frac{\partial f_i^1}{\partial t_1}\right) = \frac{-f_i^2}{\tau}, \tag{B.26}$$

is derived. Again, multiplying by $\vec{c_i}$ and summing over all i we get

$$\rho\frac{\partial u}{\partial t_2} + (1 - \frac{1}{2\tau})\left(\sum_i \vec{c_i}\vec{c_i}\nabla f_i^1\right) = 0 \tag{B.27}$$

This then implies that

$$\rho\frac{\partial u}{\partial t_2} + (1 - \frac{1}{2\tau})\left(\begin{array}{c} \nabla\left(\begin{array}{c}\sum_i c_{ix}c_{ix}f_i^1 \\ \sum_i c_{ix}c_{iy}f_i^1\end{array}\right) \\ \nabla\left(\begin{array}{c}\sum_i c_{iy}c_{ix}f_i^1 \\ \sum_i c_{iy}c_{iy}f_i^1\end{array}\right)\end{array}\right) = 0. \tag{B.28}$$

Expressing f_i^1 as the sum of a partial time and spatial derivative using Equation B.11,

$$\sum_i c_{ix}c_{ix}f_i^1 = -\tau\left(\frac{\partial \sum c_{ix}c_{ix}f_i^e}{\partial t_1} + \frac{\partial \sum c_{ix}^3 f_i^e}{\partial x} + \frac{\partial \sum c_{iy}c_{ix}^2 f_i^e}{\partial y}\right) \tag{B.29}$$

and

$$\sum_i c_{ix}c_{iy}f_i^1 = -\tau\left(\frac{\partial \sum c_{ix}c_{iy}f_i^e}{\partial t_1} + \frac{\partial \sum c_{ix}^2 c_{iy} f_i^e}{\partial x} + \frac{\partial \sum c_{iy}^2 c_{ix} f_i^e}{\partial y}\right). \tag{B.30}$$

The assumption is made that the time derivative of the equilibrium is approximately zero. Again using Equation 3.19 for the equilibrium distribution,

$$\sum_i c_{ix}c_{ix}f_i^1 = -\tau\left(\rho\frac{\partial u_x}{\partial x} + \frac{\rho}{3}\frac{\partial u_y}{\partial y}\right), \qquad \sum_i c_{ix}c_{iy}f_i^1 = -\tau\left(\frac{\rho}{3}\frac{\partial u_y}{\partial x} + \frac{\rho}{3}\frac{\partial u_y}{\partial y}\right). \tag{B.31}$$

For the x co-ordinate of Equation B.28 becomes

$$\rho\frac{\partial u_x}{\partial t_2} + \rho(\frac{1}{2} - \tau)\left(\frac{\partial^2 u_x}{\partial x^2} + \frac{2}{3}\frac{\partial u_y}{\partial y \partial x} + \frac{1}{3}\frac{\partial^2 u_y}{\partial y^2}\right) = 0. \tag{B.32}$$

Therefore

$$\frac{\partial u_x}{\partial t_2} + (\frac{1}{2} - \tau)\left(\frac{1}{3}\frac{\partial^2 u_x}{\partial x^2} + \frac{2}{3}\frac{\partial}{\partial x}\left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y}\right) + \frac{1}{3}\frac{\partial^2 u_y}{\partial y^2}\right) = 0. \tag{B.33}$$

From the zero divergence rule, equation B.14, the middle term vanishes giving

$$\frac{\partial u_x}{\partial t_2} + (\frac{1}{2} - \tau)\left(\frac{1}{3}\frac{\partial^2 u_x}{\partial x^2} + \frac{1}{3}\frac{\partial^2 u_y}{\partial y^2}\right) = 0. \tag{B.34}$$

Combining the two scales for the time derivatives we get the final Navier stokes equations,

$$\rho\frac{\partial u_x}{\partial t_1} + \frac{\partial P}{\partial x} + \rho u_x\frac{\partial u_x}{\partial x} + \rho u_y\frac{\partial u_x}{\partial y} - v\left(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2}\right) = 0, \tag{B.35}$$

University of Cape Town

with $v = \frac{1}{3}(\tau - \frac{1}{2})$. This shows that the relaxation parameter, which governs how fast the distribution functions relax towards equilibrium, controls the viscosity of the fluid being simulated. Had time and space steps not been normalised to 1, the following expression would have been retrieved: [Wolf-Gladrow, 2000]

$$v = \frac{c^2 \Delta t}{3} \left(\tau - \frac{1}{2} \right). \tag{B.36}$$

University of Cape Town

Appendix C

Single CPU Profiles

C.1 TAU profile

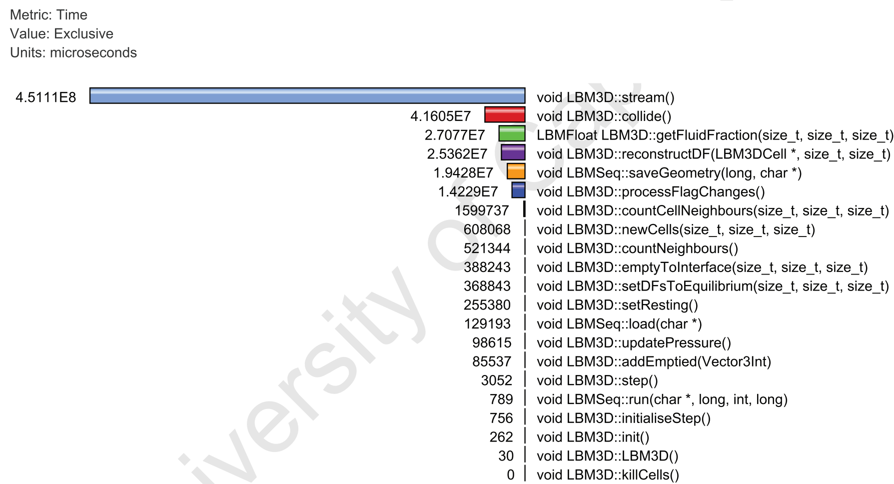


Figure C.1: A profile of the single CPU implementation.
The majority of the time of the fluid simulation is spent in the stream and collide function, which performs the numerical simulation and the mass tracking.

University of Cape Town

C.2 Operation Counts

	Memory Acceses	sqrt	Floating point	Integer operations
Fluid	574	2	401	144
Stream	38	0	0	0
Check neighbour type	19		0	0
Copy appropriate DF	19		0	0
Collide	536	2	401	144
Check for sink cells	1		0	0
Zero current pressure and velocity	1		0	0
Check for override velocity	1		0	0
Calculate current pressure	19		0	19
Calculate current velocity	133		57	57
Current velocity / Pressure			1	0
add gravity force	3		0	3
Save velocity and pressure	3		0	3
Check for fill or empty status	2		0	0
Stability correction	335	2	305	43
Lattice Vector dot current velocity	57		57	0
Calculate Equilibrium-DF	57		61	3
Stress Tensor	216		180	36
Calculate Smagorinsky constant	4	2	4	2
Calculate relaxation parameter	1		3	2
Relax DFs to equilibrium	38		38	19
Interface	817	0	538	232
Fetch neighbour types	2		0	0
Calculate Cell Normal	15	0	3	3
Calculation	3		3	3
Get Fluid Fraction (6)	12		0	0
Stream	243	0	127	82
Check neighbour type	19		0	0
Copy appropiate DF	19		0	0
Normal Dot velocity vector	7		3	0
Neighbour type (quantity):			0	0
Fluid (5)	15	0	0	10
Mass exchange	3		0	2
Interface(8)	68	0	24	32
Get destination neighbour types	2		0	0
Calculate average mass	2		2	2
Mass exchange	4.5		1	2
Empty (5)	115	0	100	40
Reconstruct DF	23		20	8
Collide	542	0	405	144
as above	536		401	144
correction for filled or empty check	6		4	0

Table C.1: The operations used during the stream and collide steps of the LBM for the still pond test case. Totals for the respective cell types are shown with a single line above and a double line below, while sub-totals have a single line below and sub-sub-totals have a single dashed line below. The still pond test case is considered, as the fluid division can be predicted.

University of Cape Town

Appendix D

Extended Tests and Results

D.1 Latency and bandwidth values for CRUNCH and CHPC

SKaMPI [Reussner, 2002]is a generic benchmarking tool for any MPI implementation. It has a number of default tests for MPI 1 and MPI 2 function sets, but is easily extended to perform specific tests or tests which have been left out of the suite. The simple test, `Pingpong_Send_Recv`, is all that is required to come up with the estimates we need. First SKaMPI finds the node with the highest latency with the root node, this node is used to perform the measurements. The `Pingpong_Send_Recv` test records sum of the time taken for the root node to send a buffer of defined size to the max latency node and for that node to send a buffer of the same size back. Recording this time for a number of different buffer sizes yields the graph shown in Figure D.1. A linear regression analysis is used to determine $T_{start-up}$ and T_{data} for various messages sizes as different protocols are used at different times. For above 64kb, CRUNCH has a start-up time of $1614\mu s$ and a bandwidth of $8.714\mu skb^{-1}$ at a R^2 value of 1, while for below 64kb, the start-up time is $520.8\mu s$ and bandwidth $14.05\mu skb^{-1}$ at a R^2 value of 0.987. For the CHPC, in region of 0-256kb, we have a start-up time of $87.06\mu s$ and bandwidth of $2.469\mu skb^{-1}$ at a R^2 value of 0.987, while in the region 256kb-1536kb a start-up time of $324.0\mu s$ and bandwidth of $1.588\mu skb^{-1}$ at a R^2 value of 0.980.

University of Cape Town

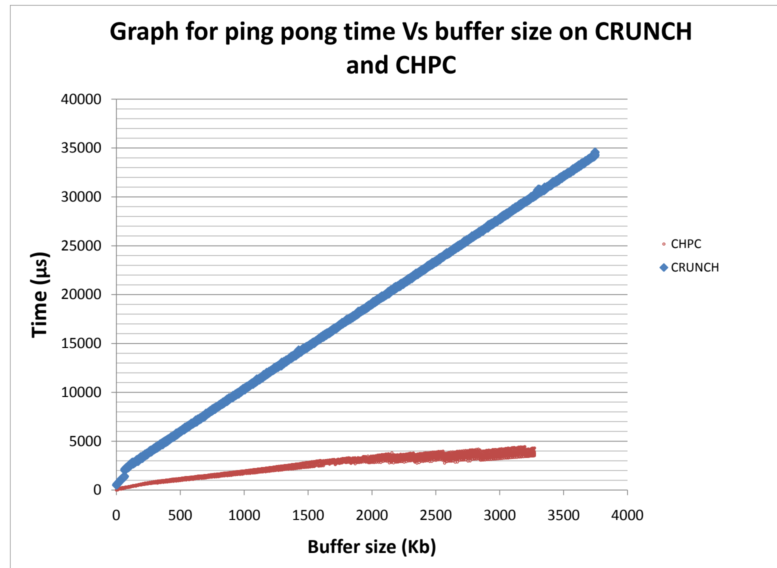
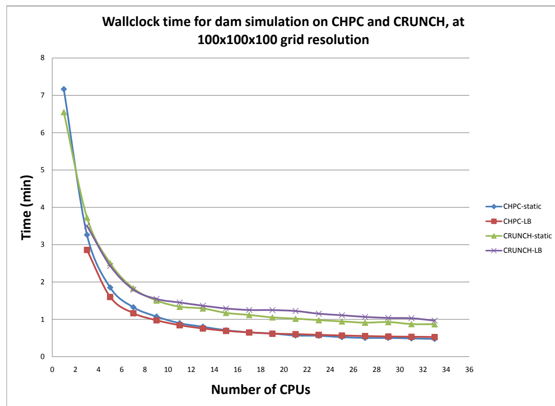
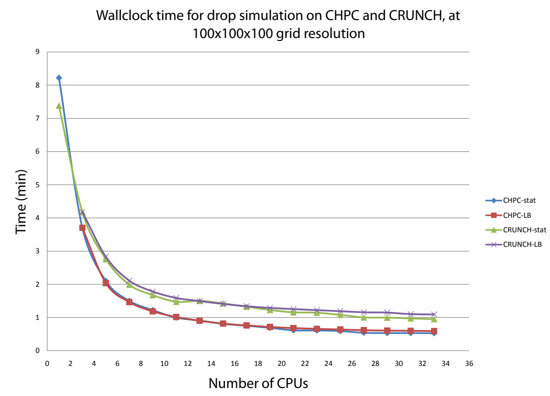


Figure D.1: The `Pingpong_send_recv` times using SKaMPI.
Shown for CRUNCH(left) and the CHPC(right) for varying buffer sizes. A noticeable jump occurs at the 64kb mark for CRUNCH due to the 64kb TCP buffer, while there are a number of different linear type effects for the CHPC. Note that the CHPC uses Infiniband, which is highly optimised, while CRUNCH uses Gigabit Ethernet.

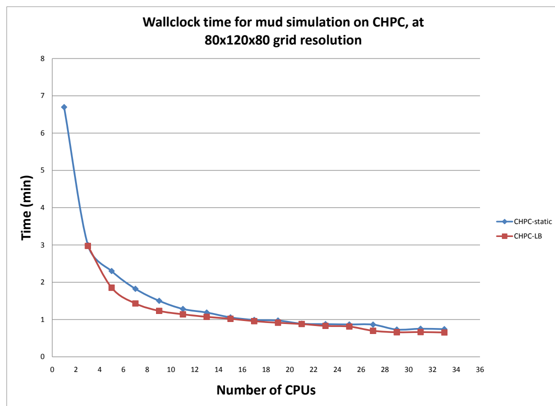
University of Cape Town



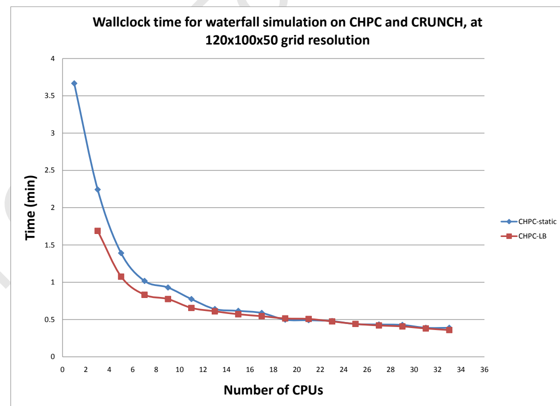
(a)



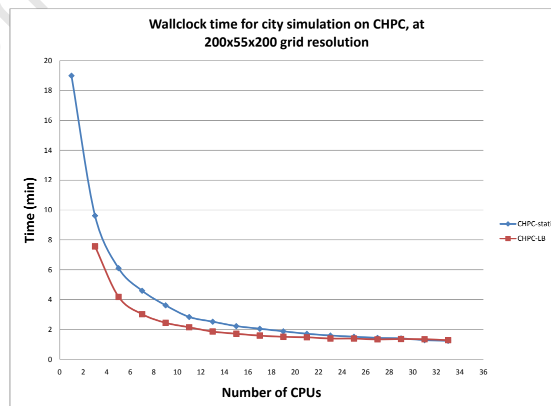
(b)



(c)

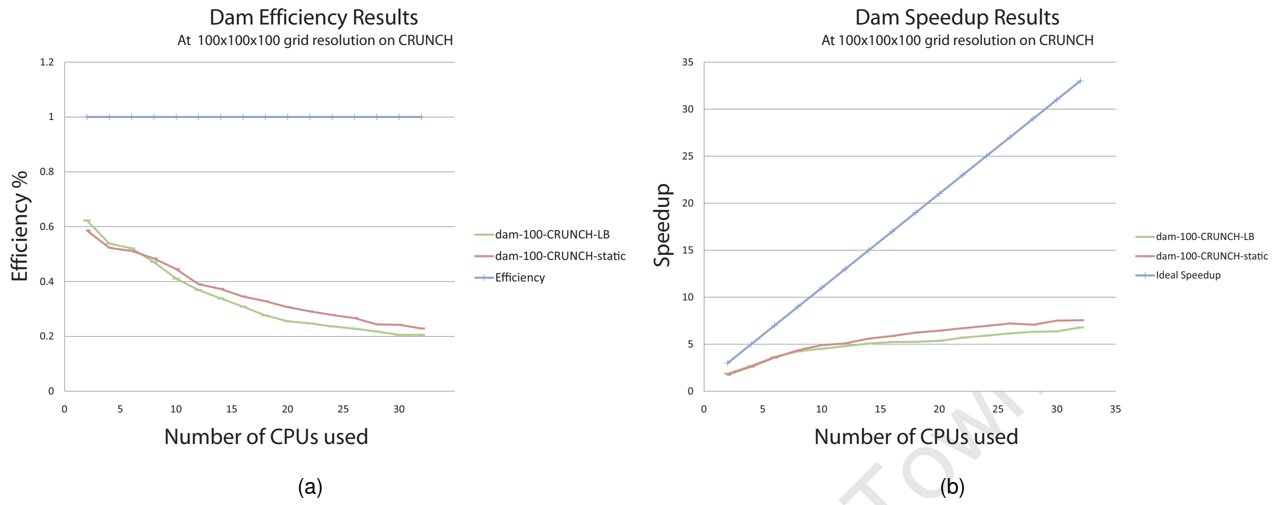


(d)



(e)

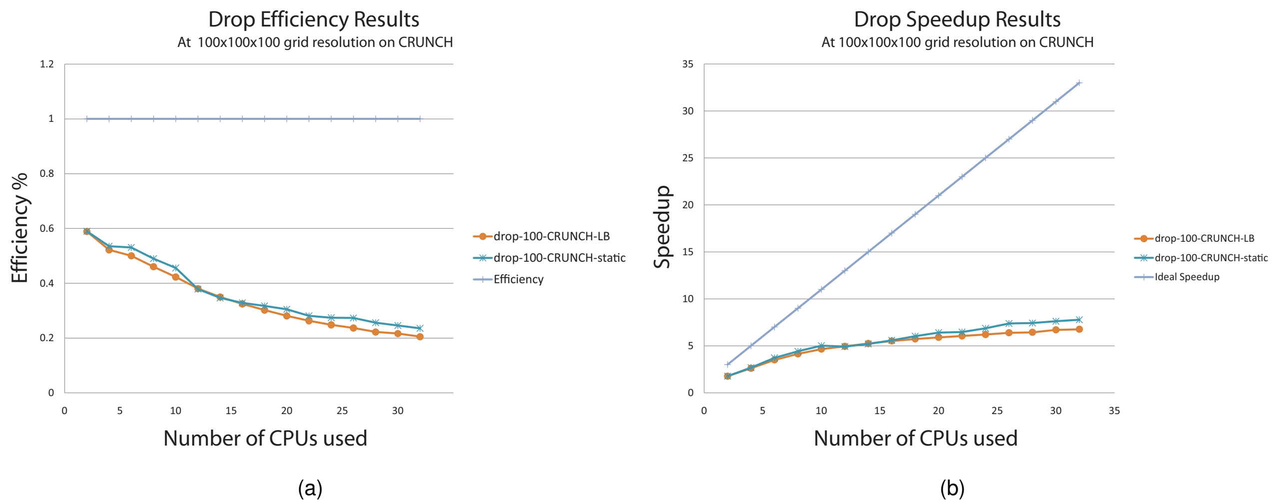
Figure D.2: Wallclock time for simulations on the CHPC and CRUNCH architectures.



(a) (b)

Figure D.3: Speedup and efficiency results for dam for a grid resolution of $100 \times 100 \times 100$.

University of Cape Town

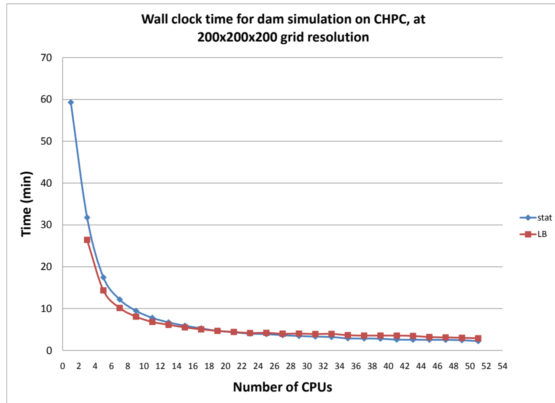


(a) (b)

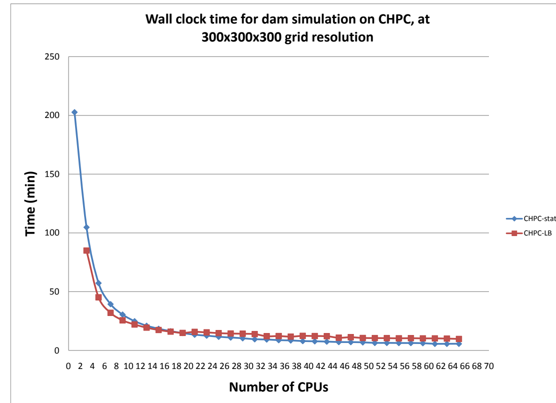
Figure D.4: Speedup and efficiency results for dam for a grid resolution of $100 \times 100 \times 100$.

D.2 Simulations times

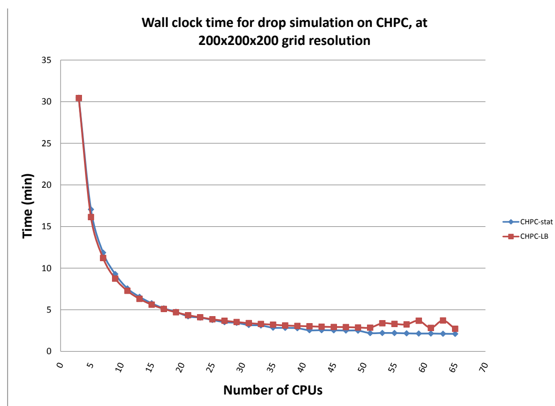
D.2.1 Wall clock times



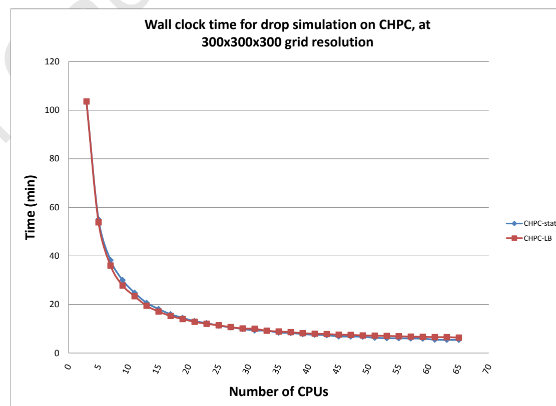
(a)



(b)



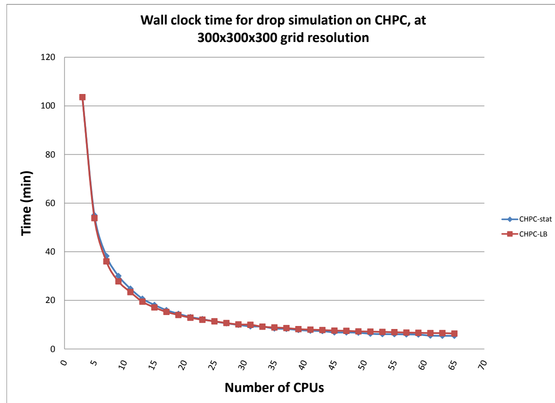
(c)



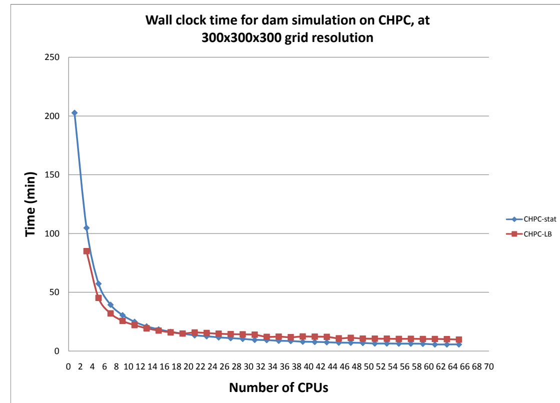
(d)

Figure D.5: Wall clock time for simulations at medium resolutions

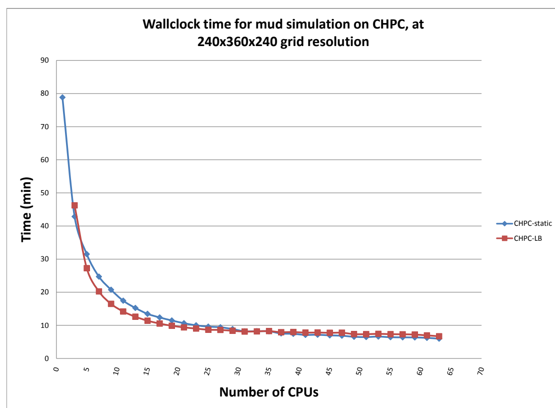
University of Cape Town



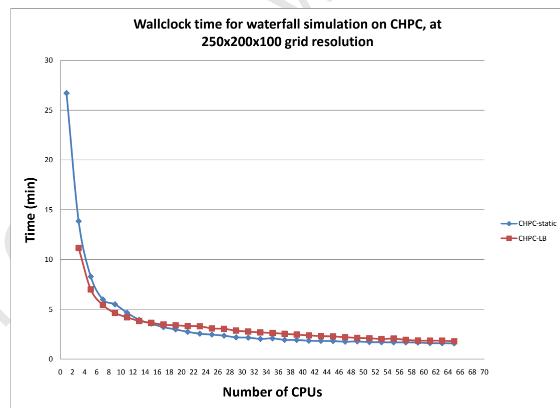
(a)



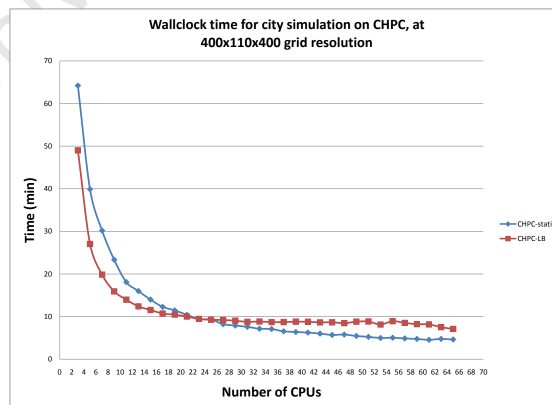
(b)



(c)



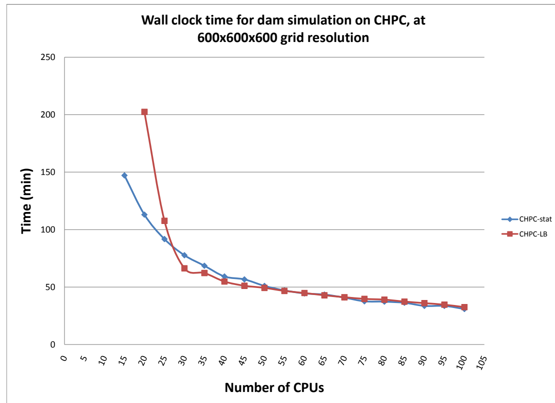
(d)



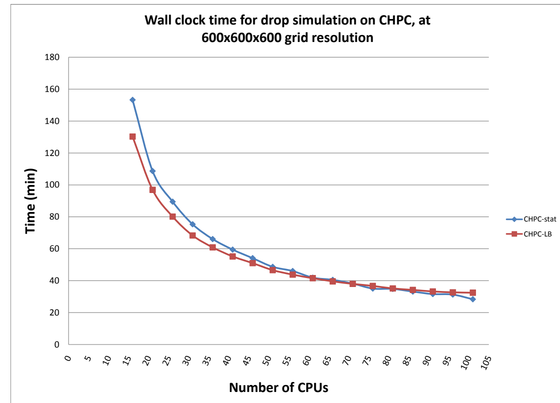
(e)

Figure D.6: Wall clock time for simulations at medium resolutions

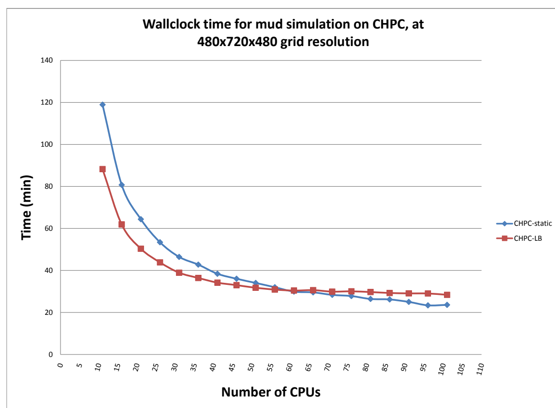
University of Cape Town



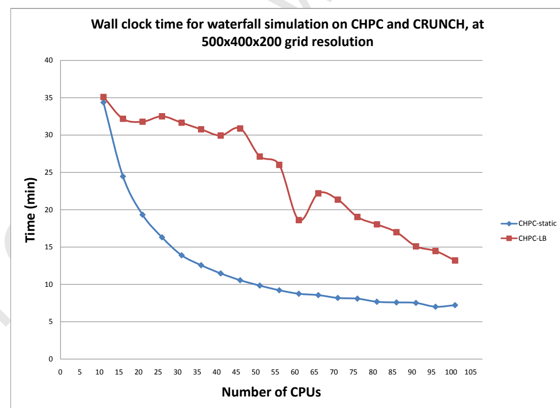
(a)



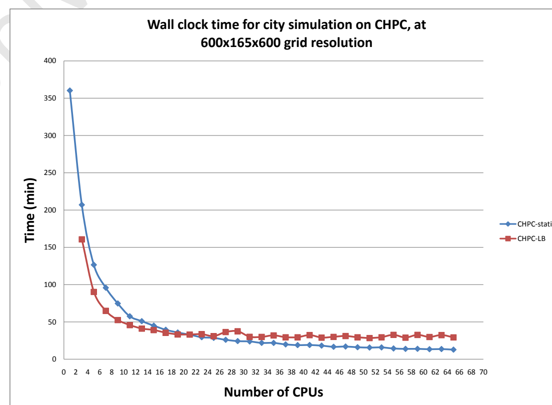
(b)



(c)



(d)



(e)

Figure D.7: Wall clock time for simulations at high resolutions

Bibliography

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. doi: 10.1142/S0129183197000746.

G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

Yukiya Aoyama and Jun Nakano. *RS/6000 SP: Practical MPI Programming*. IBM, Austin, Texas, 1999.

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: http://doi.acm.org/10.1145/125826.125925.

Adam W. Bargteil, Tolga G. Goktekin, James F. O'Brien, and John A. Strain. A semi-lagrangian contouring method for fluid simulation. *ACM Transactions on Graphics*, 25(1), 2006.

GK Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

Glen L. Beane. The effects of microprocessor architecture on speedup in distrbuted memory supercomputers. Master's thesis, The Graduate School The University of Maine, 2004.

M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, 1987. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.1987.1676942.

P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94(3):511–525, May 1954. doi: 10.1103/PhysRev.94.511.

Robert Bridson, Ronald Fedkiw, and Matthias Müller-Fischer. Fluid simulation: Siggraph 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 1–87, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-364-6. doi: http://doi.acm.org/10.1145/1185657.1185730.

Deborah Carlson. Wave displacement effects for surf's up. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 91, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1278780.1278890.

Mark Carlson, Peter J. Mucha, III R. Brooks Van Horn, and Greg Turk. Melting and flowing. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 167–174, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-573-4. doi: http://doi.acm.org/10.1145/545261.545289.

University of Cape Town

Mark Carlson, Peter J. Mucha, and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3):377–384, 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015733.

M.T. Carlson. *RIGID, MELTING, AND FLOWING FLUID*. PhD thesis, Georgia Institute of Technology, 2004.

Jim X. Chen and Niels 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. ISSN 1077-3169. doi: http://dx.doi.org/10.1006/gmip.1995.1012.

S. Chen and G. D. Doolen. Lattice Boltzmann Method for Fluid Flows. *Annual Review of Fluid Mechanics*, 30: 329–364, 1998. doi: 10.1146/annurev.fluid.30.1.329.

Nuttapong Chentanez, Bryan E. Feldman, François Labelle, James F. O'Brien, and Jonathan R. Shewchuk. Liquid simulation on lattice-based tetrahedral meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-624-4.

Holger Dachsel, Michael Hofmann, and Gudula 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. ISBN 978-3-540-74465-8. doi: 10.1007/978-3-540-74466-5_73. URL http://www.springerlink.com/content/avqn88571rt4t4qp/.

P.G. de Gennes, F. Brochard-Wyart, and D. Quéré. *Capillarity and Wetting Phenomena: Drops, Bubbles, Pearls, Waves*. Springer, 2004.

J.-C. Desplat, I. Pagonabarraga, and P. Bladon. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. *Computer Physics Communications*, 134:273–290, March 2001. doi: 10.1016/S0010-4655(00)00205-8.

Dominique d'Humières, Irina Ginzburg, Manfred Krafczyk, Pierre Lallemand, and Li-Shi Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Phil. Trans. R. Soc. A*, 360:437–451, 2002.

Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-55860-871-0.

Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566645.

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.

Raanan Fattal and Dani Lischinski. Target-driven smoke animation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 441–448, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1186562.1015743.

Ronald Fedkiw, Jos Stam, and Henrik Wann 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. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383260.

University of Cape Town

Markus Feilner. *OpenVPN: Building and Integrating Virtual Private Networks*. Springer-Verlag, New York, 1990.

Bryan E. Feldman, James F. O'Brien, and Bryan M. Klingner. Animating gases with hybrid meshes. *ACM Trans. Graph.*, 24(3):904–909, 2005a. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1073204.1073281.

Bryan E. Feldman, James F. O'Brien, Bryan M. Klingner, and Tolga G. Goktekin. Fluids in deforming meshes. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 255–259, New York, NY, USA, 2005b. ACM. ISBN 1-7695-2270-X. doi: http://doi.acm.org/10.1145/1073368.1073405.

M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9): 948–960, 1972.

J.D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1995.

Nick Foster and Ronald 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. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383261.

Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graph. Models Image Process.*, 58(5):471–483, 1996. ISSN 1077-3169. doi: http://dx.doi.org/10.1006/gmip.1996.0039.

Nick Foster and Dimitris Metaxas. Controlling fluid animation. In *CGI '97: Proceedings of the 1997 Conference on Computer Graphics International*, page 178, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7825-9.

Alain Fournier and William T. Reeves. A simple model of ocean waves. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 75–84, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: http://doi.acm.org/10.1145/15922.15894.

U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56:1505–1508, April 1986.

Uriel Frisch, Dominique dHumi'eres, Brosl Hasslacher, Pierre Lallemand, Yves Pomeau, and Jean-Pierre Rivert. Lattice gas hydrodynamics in two and three dimensions. *Complex Systems*, 1:649–707, 1987.

Richard M. Fujimoto. Parallel simulation: parallel and distributed simulation systems. In *WSC '01: Proceedings of the 33nd conference on Winter simulation*, pages 147–157, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7309-X.

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

S. Geller, M. Krafczyk, J. Tölke, S. Turek, and J. Hron. Benchmark computations based on lattice-Boltzmann, finite element and finite volume methods for laminar flows. *Computers and Fluids*, 35(8-9):888–897, 2006.

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

University of Cape Town

Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Trans. Graph.*, 24(3):973–981, 2005. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1073204.1073299.

J. Hardy, O. de Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Phys. Rev. A*, 13:1949–1961, May 1976. doi: 10.1103/PhysRevA.13.1949.

Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965a. doi: http://scitation.aip.org/jhtml/doi.jsp.

Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965b. doi: 10.1063/1.1761178. URL http://link.aip.org/link/?PFL/8/2182/1.

Xiaoyi He and Li-Shi Luo. A priori derivation of the lattice boltzmann equation. *Phys. Rev. E*, 55(6):R6333–R6336, Jun 1997. doi: 10.1103/PhysRevE.55.R6333.

N.J. Higham. *Accuracy and stability of numerical algorithms*. Siam, 2002.

F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *Europhysics Letters*, 9:663–+, August 1989.

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

S. Hou, J. Sterling, S. Chen, and G. D. Doolen. A Lattice Boltzmann Subgrid Model for High Reynolds Number Flows. *Contributions to Mineralogy and Petrology*, pages 1004–+, January 1994.

Lee Howes and David Thomas. Real time simulation and rendering of 3d fluids. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald 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. ISBN 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179352.1141959.

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. doi: 10.1016/S0010-4655(98)00025-3.

George Karypis and Vipin Kumar. Multilevel k -way hypergraph partitioning. In *Design Automation Conference*, pages 343–348, 1999. URL citeseer.ist.psu.edu/article/karypis98multilevel.html.

Michael Kass and Gavin 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. ISBN 0-201-50933-4. doi: http://doi.acm.org/10.1145/97879.97884.

Richard Keiser, Bart Adams, Dominique Gasser, Paolo Bazzi1 Philip Dutrï¿½, and Markus Gross. A unified lagrangian approach to solid-fluid animation. *Point-Based Graphics*, 20(21):125–148, 2005.

Richard Keiser, Bart Adams, Leonidas J. Guibas, Philip Dutri, and Mark Pauly. Multiresolution particle-based fluids. Technical report, ETH CS, 2006.

University of Cape Town

Bryan M. Klingner, Bryan E. Feldman, Nuttapong Chentanez, and James F. O'Brien. Fluid animation with dynamic meshes. In *Proceedings of ACM SIGGRAPH 2006*, pages 820–825, August 2006.

C. Körner and R. F. Singer. Processing of metal foams| -| challenges and opportunities. *Advanced Engineering Materials*, 2(4):159–165, 2000. ISSN 1527-2648.

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

Carolin Körner, Thomas Pohl, , Nils Thürey, and Thomas Zeiser. *Parallel Lattice Boltzmann Methods for CFD Applications*, volume 51. Springer Berlin Heidelberg, 2006.

Géraud Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127, New York, NY, USA, 2003. ACM. ISBN 1-58113-661-7. doi: http://doi.acm.org/10.1145/777412.777433.

W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7):444–456, 2003.

A. Lindenmayer. Adding Continuous Components to L-Systems. *Lecture Notes In Computer Science; Vol. 15*, pages 53–68, 1974.

Mats Lindh. Marching cubes implementation, March 2003. http://www.ia.hiof.no/~borres/cgraph/explain/marching/p-march.html.

William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/37402.37422.

F. Losasso, G. Irving, E. Guendelman, and R. Fedkiw. Melting and Burning Solids into Liquids and Gases. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, pages 343–352, 2006a.

Frank Losasso, Frédéric Gibou, and Ron 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. doi: http://doi.acm.org/10.1145/1186562.1015745.

Frank Losasso, Tamar Shinar, Andrew Selle, and Ronald Fedkiw. Multiple interacting liquids. *ACM Trans. Graph.*, 25(3):812–819, 2006b. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1141911.1141960.

Frank Losasso, Jerry Talton, Nipun Kwatra, and Ronald Fedkiw. Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2008.37.

Hai Mao and Yee-Hong Yang. Particle-based immiscible fluid-fluid collision. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 49–55, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society. ISBN 1-56881-308-2.

Guy R. McNamara and Gianluigi Zanetti. Use of the boltzmann equation to simulate lattice-gas automata. *Phys. Rev. Lett.*, 61(20):2332–2335, Nov 1988. doi: 10.1103/PhysRevLett.61.2332.

Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics*, 13(3):305–309, 1989. URL citeseer.ist.psu.edu/miller89globular.html.

University of Cape Town

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.

C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *Proceedings of the conference on Visualization'94*, pages 281–287. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.

Dirk Muders. *Three-Dimensional Parallel Lattice Boltzmann Hydrodynamic Simulations of Turbulent Flows in Interstellar Dark Clouds*. PhD thesis, Rheinischen Friedrich-Wilhelms-Universitat, 1995.

Matthias Müller, David Charypar, and Markus 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. ISBN 1-58113-659-5.

Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 237–244, New York, NY, USA, 2005. ACM Press. ISBN 1-7695-2270-X. doi: http://doi.acm.org/10.1145/1073368.1073402.

Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566643.

P. Ning and J. Bloomenthal. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, 13(6):33–41, 1993.

Karonis N.T. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63:551–563(13), May 2003. URL http://www.ingentaconnect.com/content/els/07437315/2003/00000063/00000005/art00002.

J. Paul Peszko. Water, water, everywhere..., 2006. www.vfxworld.com.

Nils Thürey Thomas Pohl and Ulrich Rüde. Hybrid Parallelization Techniques for Lattice Boltzmann Free Surface Flows. *International Conference on Parallel Computational Fluid Dynamics*, May 2007.

TJ Poinsot and SK Lele. Boundary conditions for direct simulations of compressible viscous flows. *Journal of computational physics(Print)*, 101(1):104–129, 1992.

Simon Premoze, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross Whitaker. Particle-based simulation of fluids. *Eurographics 2003*, 22(3), 2003.

W.H. Press, B. FLANNERY, S.A. TEUKOLSKY, W.T. VETTERLING, et al. *Numerical recipes*. Cambridge University Press New York, 1986.

Y. H. Qian, D. D'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters*, 17:479–+, February 1992.

RM Ramanathan, P. Contributors, R. Curry, S. Chennupaty, R.L. Cross, S. Kuo, and M.J. Buxton. Extending the World's Most Popular Processor Architecture. *Technology intel Magazine*, 2006.

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

University of Cape Town

M. Roy, S. Foufou, and F. Truchetet. Mesh Comparison Using Attribute Deviation Metric. *International Journal of Image and Graphics*, 4(1):127–140, 2004.

Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Trans. Graph.*, 24(3):910–914, 2005. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1073204.1073282.

Maddu Shankar and S. Sundar. Asymptotic analysis of extrapolation boundary conditions for lbm. *Comput. Math. Appl.*, 57(8):1313–1323, 2009. ISSN 0898-1221. doi: http://dx.doi.org/10.1016/j.camwa.2009.01.018.

Lin Shi and Yizhou Yu. Taming liquids for rapidly changing targets. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 229–236, New York, NY, USA, 2005. ACM Press. ISBN 1-7695-2270-X. doi: http://doi.acm.org/10.1145/1073368.1073401.

Samuel Silva, Joaquim Madeira, and Beatriz Sousa Santos. Polymeco " a polygonal mesh comparison tool. In *IV '05: Proceedings of the Ninth International Conference on Information Visualisation*, pages 842–847, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2397-8. doi: http://dx.doi.org/10.1109/IV.2005.98.

Jos 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. ISBN 0-201-48560-5. doi: http://doi.acm.org/10.1145/311535.311548.

Markus Sturmer, Jan Gotz, Gregor 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.

Sauro Succi. *The Lattice Boltzmann Euqation for fluid dynamics and beyond*. Oxford University Press, New York, 2001.

Mark Sussman. A second order coupled level set and volume-of-fluid method for computing growth and collapse of vapor bubbles. *J. Comput. Phys.*, 187(1):110–136, 2003. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/S0021-9991(03)00087-1.

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

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, 2005a.

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, 2005b.

N. Thurey, K. Iglberger, and U. Rude. Free surface flows with moving and deforming objects with LBM. In *Vision, Modeling, and Visualization*, 2006.

N. Thürey, R. Keiser, and M. Pauly Ulrich Rüde. Detail-preserving fluid control. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 7–12, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN 3-905673-34-7.

University of Cape Town

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.

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

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

Nils Thürey and Ulrich Rüde. Optimized free surface fluids on adaptive grids with the lattice boltzmann method. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Posters*, page 112, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1186954.1187082.

Nils Thürey and Ulrich Rüde. "stable free surface flows with the lattice boltzmann method on adaptively coarsened grids". *Computing and Visualization in Science*, 2008.

Stephan Trojansky, Thomas Ganshorn, and Oliver Pilarski. 300's liquid battlefield: fluid simulation spartan style. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 95, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1278780.1278894.

T. N. Venkatesh, V. R. Sarasamma, Rajalakshmy S., Kirti Chandra Sahu, and Rama Govindarajan. Super-linear speed-up of a parallel multigrid Navier-Stokes solver on Flosolver. *RESEARCH ARTICLES*, 88:589–593, 2004.

Junye Wang, Xiaoxian Zhang, Anthony G. Bengough, and John 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. doi: 10.1103/PhysRevE.72.016706. URL http://link.aps.org/abstract/PRE/v72/e016706.

Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie 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. ISBN 1-58113-659-5.

Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004a. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2004.1260768.

Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004b. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2004.1260768.

Jens Wilke, Thomas Pohl, Markus Kowarschik, and Ulrich Ruede. Cache performance optimizations for parallel lattice boltzmann codes. In *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 441–450. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40788-1. doi: 10.1007/b12024. URL http://www.springerlink.com/content/p7jagg6fjvt4ryec/.

B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and lattice Boltzmann Models*. Springer-Verlag, New York, 2000.

Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24:965–972, 2005.

University of Cape Town