Realtime Computational Fluid Dynamics Simulations using the Lattice Boltzmann Method
TJHSST Senior Research Project
Computer Systems Lab 2009-2010

Thomas Georgiou

June 15, 2010

Abstract

Fluid simulations are useful in many different areas ranging from weather modeling to microscopic physics. Using the conventional method of solving the districtized Navier-Stokes equations is very computationally intensive and relatively hard to parallelize. The lattice boltzmann method instead uses the discrete Boltzmann equation to simulate Newtonian fluids using various collision models.

Keywords: computational fluid dynamics, lattice boltzmann methods, parallel computing

Contents

1 Introduction

Fluid dynamics are useful in a broad range of fields including meteorology, computer graphics, aerodynamics, and microscopic physics. The purpose of this project is to accelerate relatively new methods in the field of computational fluid dynamics in order to be able to run realtime simulations. This includes using new methods that can be parallelized more effectively and vectorizing these methods and running them on new hardware using GPGPU techniques. Historically developed from a cellular automata approach, the Lattice Boltzmann method provides a scalable simulation method that is rapidly becoming popular.

2 Background

2.1 Lattice Gas Cellular Automata

Historically, the Lattice Boltzmann Method has evolved from the Lattice Gas Cellular Automata (LGCA) Method. This approach to fluid dynamics modelling is to make a hexagonal grid, with every grid point having a set of 7 possible velocities, each pointing to the neighboring lattice point, or staying still. No two particles can occupy the same point with the same velocity. At each time step, particles move to the next point dictated by their velocity. If another particle is also moving to the same space, a collision model is used to determine where each particle settles. From this microscopic model, macroscopic behavior consistent with the Navier-Stokes equations emerges. However, this method also has many drawbacks including statistical noise, lack of a range of physical parameters, and difficulties in three dimensions. The Lattice Boltzmann Method was created to overcome this, mainly by replacing the boolean particle number in every lattice direction with a floating point average, or distribution function.

2.2 Boltzmann Equation

The Lattice Boltzmann Method can be obtained from the discrete velocity model of LCGA or from districtizing the Boltzmann equation. Here, the derivation from the Boltzmann equation is presented.

The Boltzmann equation

$$f(x + vdt, v, t + dt) = f(x, v, t) + \Omega(x, v, t)$$

describes the time evolution of system of particles that interact with each other via the collision operator Ω. It consists of two parts, streaming and collisions. During streaming, particles are moved according to their velocities. During the collision stage, the distribution functions (DFs) at each lattice point undergo a collision operator, which is left as a choice.

2.3 The BGK Collision Operator

A simple and popular collision operator is the Bhatnagar, Gross, and Krook (BGK) collision operator [2].

$$\Omega_{BGK} = -\frac{f - f_eq}{\tau}$$

This is the single-time-relaxation model where collisions tend to push the system towards local equilibrium. This model is computationally simple, relying only on the local distribution functions, yet accurate, making it ideal, and thus very popular, for use in lattice Boltzmann Simulations. It is the one used in this simulation, yielding

$$f(x + vdt, v, t + dt) = f(x, v, t) - \frac{f - f_eq}{\tau}$$

2.4 Discretization of Phase Space

In order to solve Boltzmann equation numerically, the domain must be districtized in phase space, consisting of time, configuration space, and velocity space. Time is split up by time step. Configuration space is split apart into a lattice with a discrete set of velocities connecting neighboring nodes. Discretely, the equation becomes

$$f_i(x + e_i, t + dt) = f_i(x, t) - \frac{f_i - f_i^{eq}}{\tau}$$

where f_i are the distribution functions at every lattice point corresponding to the velocity vectors e_i where $i = 1..m$. Lattices are classified by a DnQm

scheme where n is the number of dimensions and m is the number of velocities. For example, D2Q9 is a two dimensional lattice with 9 velocities connecting neighboring nodes (4 to each corner, 4 to each midpoint, and 1 stationary) The D3Q19 lattice is a three dimensional lattice with 19 velocities connecting the neighboring nodes.

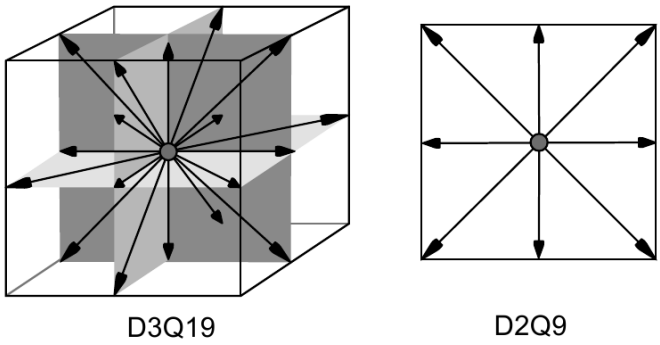


Figure 1: Various lattice and velocity configurations

3 Implementation Details

Currently, both D2Q9 and D3Q19 simulations are implemented, both using the BGK collision operator. They is programmed in the C programming language as this is a very performance intensive project and I am more comfortable in C than in Fortran. OpenGL is used to provide visualization display and input. For visualization, a grayscale image is presented with each pixel taking on the value proportional to the amount of fluid present at a lattice point. Mouse presses currently add stationary fluid at the pointer location. OpenMP is used for intra-node parallelism and MPI is used for inter-node parallelism.

Two steps are performed at each time step: the stream step and the collision step.

3.1 The Stream Step

In the stream step, the first part of the Boltzmann equation is computed. The distribution functions for each velocity at each lattice point are moved to neighboring lattice points based on their velocity.

$$f(x + e_i, e_i, t + dt) = f(x, e_i, t)$$

If a distribution function is hitting a boundary, then it undergoes the no-

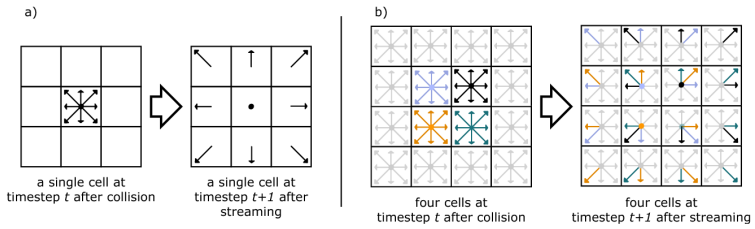


Figure 2: The stream step

slip boundary condition and stays at the same lattice point, except with the inverse velocity.

$$f(x, e_{\bar{i}}, t + dt) = f(x, e_i, t)$$

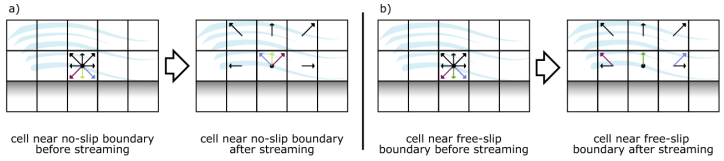


Figure 3: Boundary conditions

This step is parallelized very easily. At each time step, the only communication that needs to occur between nodes is the transfer of the status of neighboring nodes.

3.1.1 Streaming Code

```
#pragma omp parallel for
for (int x = 0; x < SIZEX; x++) {
    for (int y = 0; y < SIZEY; y++) {
  if (flags[x][y] == OBSTACLE) {
      continue;
  }
  for (int d = 0; d < DIRS; d++) {
      int nx = x + dx[d];
      int ny = y + dy[d];
      if (flags[nx][ny] == OBSTACLE) {
    //no slip boundary condition
    domain[new][x][y][I(d)] += domain[old][x][y][d];
      } else {
    domain[new][nx][ny][d] = domain[old][x][y][d];
      }
  }
    }
}
```

3.2 The Collision Step

In the collision step, the second part of the Boltzmann equation is computed: the interactions that particles have with each other. The BGK collision operator that is used is based off the fact that collisions tend to make the particles approach equilibrium, governed by the Maxwell-Boltzmann distribution. At each time step, a finite number of collisions occur, so the particles are only pushed partway towards equilibrium. So the particle distribution functions after collisions are a mixture of the pre-collision distribution functions and equilibirum distribution functions.

The equilibrium distribution is found by taking the low Mach number expansion of the Maxwell-Boltzmann distribution

$$\sqrt{\frac{m}{2\pi kT}} e^{\frac{-mv^2}{2kT}}$$

yielding

$$f_i^{eq} = w_i(\rho + 3e_i \cdot u - \frac{3}{2}u^2 + \frac{9}{2}(e_i \cdot u)^2)$$

where

$$\rho = \sum f_i$$

$$u = \sum e_i f_i$$

After computing the equilibrium distribution, the distribution functions are then relaxed using the following relationship.

$$f(x, e_i, t + dt) = (1 - \omega)f(x, e_i, t + dt) + \omega f_i^{eq}$$

where ω is the parameter controlling the viscosity of the fluid. Values close to 0 represent very viscous flows. Omega is determined from the viscosity via the following relation

$$\omega = \frac{2}{6v + 1}$$

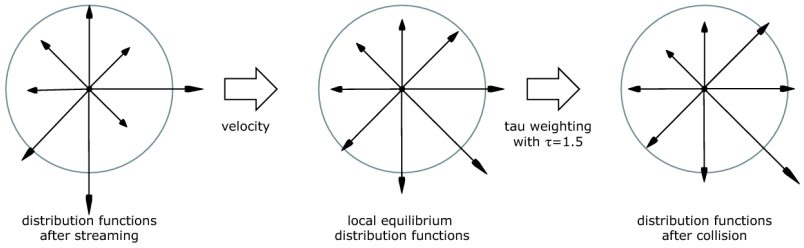


Figure 4: The collision step

3.2.1 Collision Code

```
#pragma omp parallel for
for (int x = 0; x < SIZEX; x++) {
  for (int y = 0; y < SIZEY; y++) {
    if (flags[x][y] == OBSTACLE)
      continue;
    float rho = 0.0;
    float ux = 0.0; float uy = 0.0;
    for (int d = 0; d < DIRS; d++) {
      rho += domain[new][x][y][d];
      ux  += dx[d] * domain[new][x][y][d];
      uy  += dy[d] * domain[new][x][y][d];
    }
    if (flags[x][y] == VELOCITY)
      ux = 0.1*rho;
    for (int d = 0; d < DIRS; d++) {
      float eq = 0.0;
```

```
            eq += rho;
            eq += 3*(dx[d]*ux+dy[d]*uy);
            eq -= (ux*ux+uy*uy)*3/2;
            eq += (dx[d]*ux+dy[d]*uy)*(dx[d]*ux+dy[d]*uy)*9/2;
            eq *= weights[d];
            domain[new][x][y][d] = (1-omega)*domain[new][x][y][d] +
                omega*eq;
        }
    }
}
```

3.3 Different Cell Types

In order to handle the different cell types such as obstacles, fluid, constant velocity, gas, and interface cells, the type of each cell must be recorded. An array the size of the domain is used for this.

3.4 Visualization

There are many methods to visualize fluids on a computer screen showing different things from density to velocity to pressure to vorticity.

3.4.1 Density Plot

One way of visualization is to plot the density at each lattice point by setting pixels to a color proportional to the amount of fluid present. This shows fluid at each point, but fails to capture any other information such as velocity when density is constant as is the case in incompressible flows such as water.

3.4.2 Tracer Particles

Non-fluid tracer particles may be placed in the fluid. These particles are then advected by the fluid at each timestep, using Euler's method or moving the particle by dt times the fluid velocity. The fluid velocity is computed by finding the moments and then dividing.

$$v = \frac{u}{\rho}$$

This shows how individual components of the fluid move and allows velocity to be gauaged even in incompressible flows.

3.4.3 Velocity Vector Field

Another way of showing velocity is to draw the velocity vector field. On an evenly spaced grid, vectors are drawn corresponding to the velocity at that grid point. One drawback to this is that it is hard to cover a wide range of velocities effectively.

3.4.4 Vorticity

Another metric used in fluid simulations is vorticity, which is the curl of the velocity field.

$$\vec{\omega} = \vec{\nabla} \times \vec{v}$$

It measures how much a the fluid is rotating around. Clockwise rotation is represented as red while counter-clockwise is represented as blue.

3.4.5 Streamlines

Streamlines are the lines that a particle would follow through a path if it were to pass through a fluid at a given time. Compared to tracer particles, streamlines show the path that would be taken at every timestep rather than having to view the tracer particles over an interval of time to see the direction the fluid is moving. Streamlines were implemented in the flow past a cylinder case by starting lines at the rightmost edge of the domain at evenly spaced intervals and then moving backwards a fixed number of times according to the velocity at that point. At every step, $\frac{dx}{dy}$ is calculated and then the streamline is moved up or down corresponding to it.

As the grid is discrete and not continuous, the streamlines would become very jagged. Bilinear interpolation was implemented to smooth them out. This consisted of linearly interpolating the velocity field in the horizontal direction at two different y values and then linearly interpolating the resulting values in the vertical direction.

3.5 Simulation File Format

A simulation file format is used in order to handle different test cases. The domain file (with suffix $.d$) records the types of cells at each gridpoint as well as the size of the simulation. It consists of three space seperated integers, SIZEX, SIZEY, and SIZEZ, and then a whitespace-delimited list of cell types in column-major order with the following ids:

0	obstacle
1	fluid cell
2	fluid interface cell
3	forced velocity cell
4	exit cell

Obstacle cells undergo no-slip bounce back collisions. Fluid cells are the basis of the simulation. Fluid interface cells provide the boundary between fluid and gas. Forced velocity cells are fluid cells but with the velocity moments forced to a fixed velocity during the collision step. Exit cells are equivalent to forced velocity cells albeit with special considerations taken to conserve mass as they are on the edge of the simulation domain.

3.6 Visualization using Paraview

Paraview is an open source scientific visualization tool develpoed by Sandia National Labs, Kitware Inc, and Los Alamos National Labs. It uses a distributed computing approach allowing it to analyze terascale datasets as well as on small laptops for smaller data.

3.6.1 Moving Data into Paraview

In order to visualize simulation output using Paraview, the simulation must output data in a format paraview can read. The simplest is the legacy VTK file format [1]. The header looks like this:

```
# vtk DataFile Version 3.0
LBM
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 128 32 64
ORIGIN 0 0 0
SPACING 1 1 1
POINT_DATA 262144
VECTORS velocity float
```

followed by a space seperated ascii floating point values describing the velocity at every grid point. Paraview can then import this data.

Polygon data is imported using the same VTK format but with a different header.

```
# vtk DataFile Version 3.0
```

```
LBM2
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 128 32 64
ORIGIN 0 0 0
SPACING 1 1 1
POINT_DATA 262144
SCALARS LBM int 1
LOOKUP_TABLE default
```

followed by either a "0" or a "1" for every grid point based on whether there is or is not an obstacle at that point.

3.6.2 Using Paraview

After importing velocity field data and polygon data into Paraview, it can be visualized in many different ways. The contour filter should be applied to the polygon data in order to get obstacles to display as polygons. Then, arrows can be used to visualize the velocity field or streamlines can be generated using the corresponding filters. The streamlines can then be colored as tubes based on the velocity of the fluid at that point.

Figure 5: Flow past a cylinder in 3D visualized using Paraview

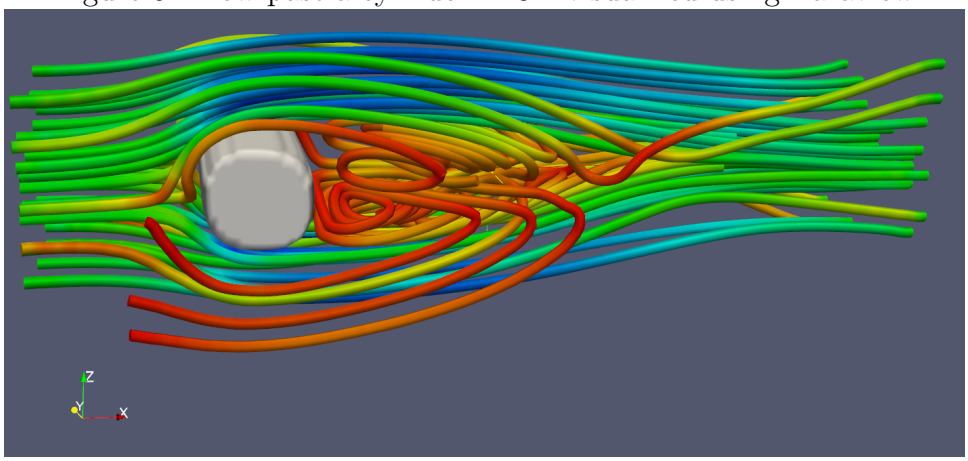
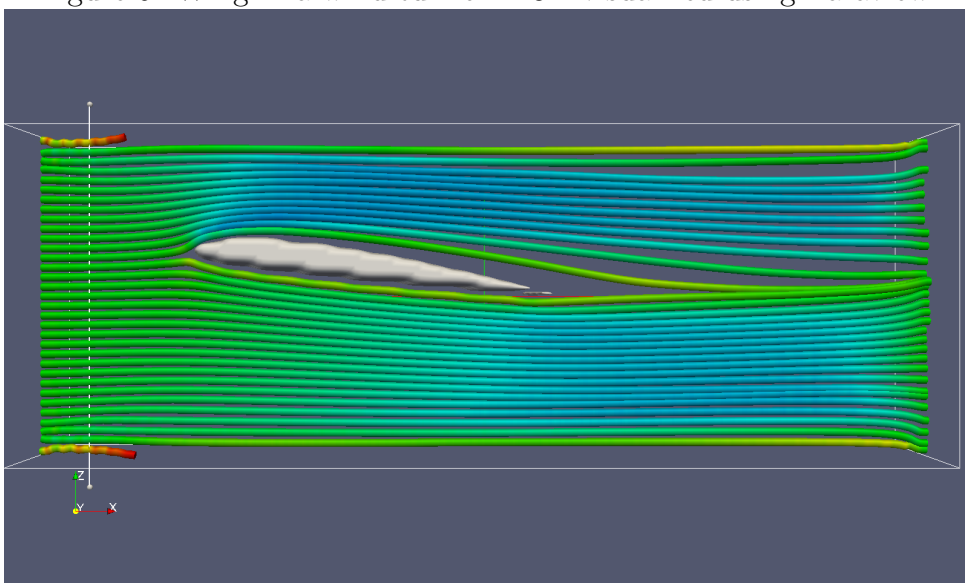


Figure 6: Wing in a wind tunnel in 3D visualized using Paraview



4 Parallelization

4.1 Shared Memory Parallelization using OpenMP

Every timestep, the 3D grid is iterated upon in the x,y, and z directions. Parallelization using OpenMP is trivial in this case. It merely involves placing *#pragma omp parallel for* before the top level for loop:

```
#pragma omp parallel for
for (int x = 0; x < SIZEX; x++) {
  for (int y = 0; y < SIZEY; y++) {
    for (int z = 0; z < SIZEZ; z++) {
    //simulation
    ...
    }
  }
}
```

4.2 Cluster Parallelization using MPI

In order to scale past a single node, the MPI programming model is used to distribute the simulation across a network. The Lattice Boltzmann Method

is well suited for this. The first step is splitting up the simulation domain for distribution into n nodes. This is done by splitting the simulation domain into n $\frac{SIZEX}{n}$ pieces. The only communication required between these pieces is then propagating the distribution functions on the edges to the neighboring pieces at every time step. The collision step requires no communication at all.

The current code is setup to run one master node which handles initializing the domain and handling display and output i/o. Then n worker nodes are run which run the simulation itself.

Further gains in performance can be seen by interleaving processing and communication. At the start of the streaming step, asynchronous MPI sends and receives are started to stream distribution functions to neighboring nodes. Then, the rest of the stream step interior to a node is processed as well as interior collision steps that do not depend on the data from neighboring nodes. Then, the processing blocks on the MPI receives, which are most likely done by this time, after which the final collision step is performed and then the process repeats again.

Every m time steps, the workers send back the simulation domain to the master node in order for visualization. Care must be taken to make m large enough as to not bottleneck the simulation on communication between the workers and the master nodes.

Within each node communicating via MPI, OpenMP parallelism is also used to take advantage of multiple processors connected via shared memory in a node.

5 Test Cases

In order to verify the physical correctness of the simulation, various published standard test cases are used.

5.1 Lid Driven Cavity

The lid driven cavity is a rectangular domain where the walls are non-slip except for the top wall, which is non-slip, but moving at a constant velocity. The simulation is setup to set the velocity of the fluid on the topmost row of pixels to 0.01 in the right direction, for all except the rightmost pixels.

Figure 7: In process lid driven cavity simulations visualized using particles

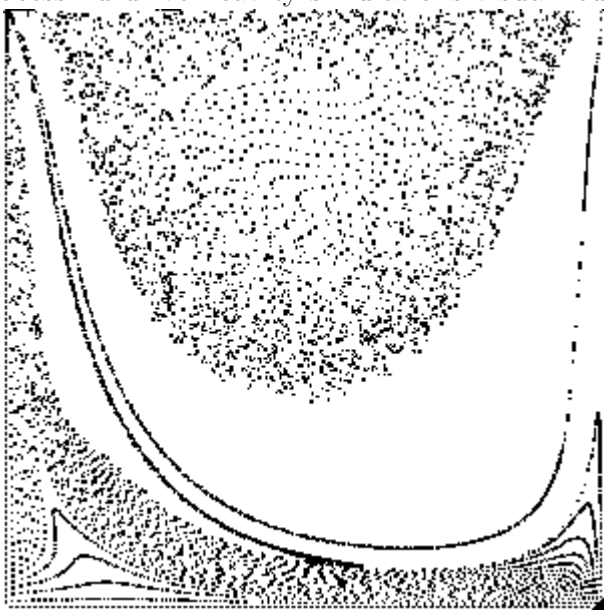
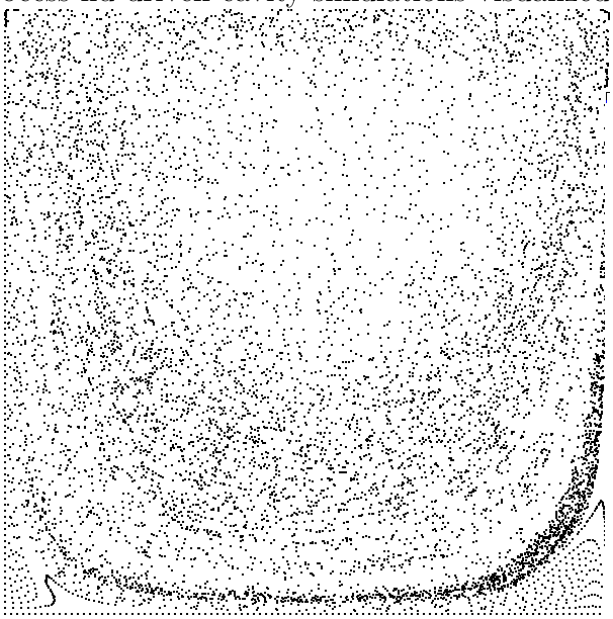


Figure 8: In process lid driven cavity simulations visualized using particles



5.2 Flow Past an Obstacle

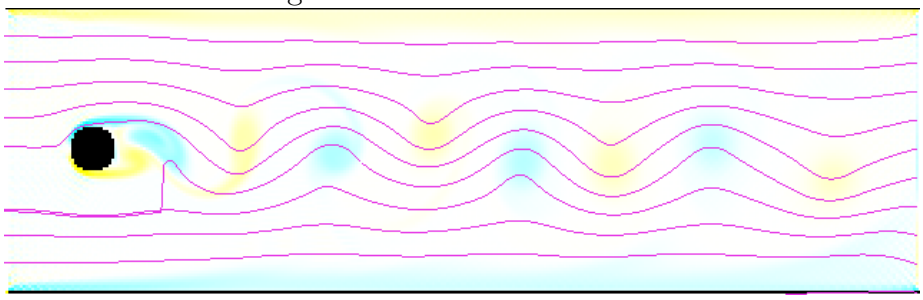
The flow past a cylinder is a rectangular domain (such as a pipe) with non-slip walls and fluid being forced through from the left and then exiting out the right.

This can also be used to simulate wind tunnel tests and compute lift and drag on arbitrary objects and airfoils.

5.2.1 Kármán Vortex Street

A Kármán vortex street is a pattern of swirling vortices created behind a blunt objects. It results in phenomena such as vibrations of car antennas or singing of suspended telephone wires.

Figure 9: In process flow past a cylinder test visualized using vorticity and streamlines demonstrating the Kármán vortex street



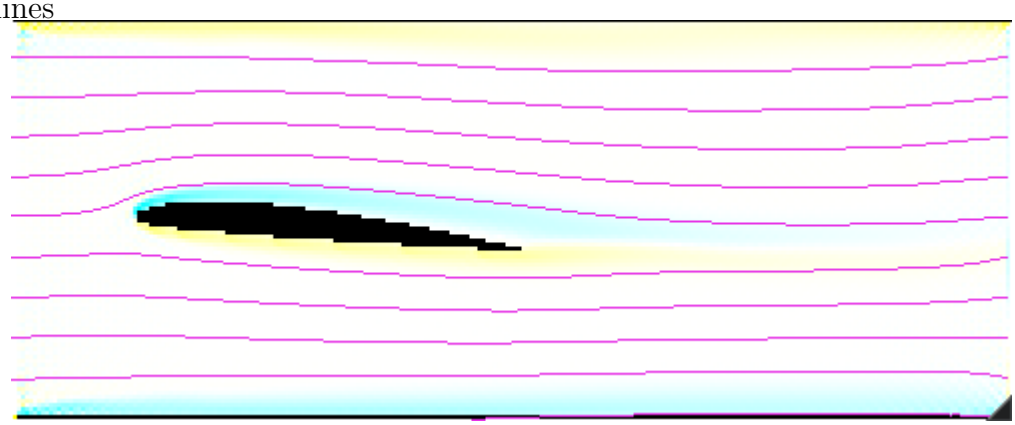
5.2.2 Lift and Drag Calculation

Wind tunnel tests can be performed by calculating the lift and drag on an obstacle. During the stream step, lift and drag are computed by summing all the impulses from the bounce back steps when an obstacle is it in the vertical and horizontal directions respectively. An exponential moving average is then used to give lift and drag numbers that are steadier with time. In parallel code, values are summed by sending lift values for sub-grids back to the master where they are then summed.

5.2.3 NACA Airfoils

The NACA series of airfoils is a standard series of airfoils that are generated by mathematical equations. The 4 digit NACA airfoil series is governed by

Figure 10: In process wind tunnel test visualized using vorticity and streamlines



the following equation:

$$y = 5tc\left[0.2969\sqrt{\frac{x}{c}} - 0.1260\left(\frac{x}{c}\right) - 0.3537\left(\frac{x}{c}\right)^2 + 0.2843\left(\frac{x}{c}\right)^3 - 0.1015\left(\frac{x}{c}\right)^4\right]$$

- c is the chord length
- x is the position of the chord from 0 to c
- y is displacement from the centerline
- t is the maximum thickness for a given length of a chord

The centerline for cambered airfoils is calculated by the following formulas

$$y_c = m\frac{x}{p^2}\left(2p - \frac{x}{c}\right)$$

from $x = 0$ to $x = pc$

$$y_c = m\frac{c-x}{(1-p)^2}\left(1 + \frac{x}{c} - 2p\right)$$

from $x = pc$ to $x = c$

To convert this to voxel data for use in the LBM simulation, iteration is performed over in the direction along the chord and then individual voxels are filled in based on the vertical values computed from the equations.

6 Expected Results

The project will be expected to yield a CFD code that is able to simulate fluids in realtime. Physically correct results should be achieved, which will be measured using fundamental laws such as conservation of mass. This can be used in realtime predictions in various fields, for example control systems dealing with fluids. The speedup techniques used can also be applied to make larger simulations run faster.

7 Results

Current results consist of a simulation of a two dimensional fluid that can be conducted in realtime on a single processor on a 300x300 grid. The simulation is currently unoptimized since I copy the memory at each timestep, which while good for simplicity and getting a simulation up and running, is terrible for performance.

A CUDA version is also implemented. Running this on a GPU is very appealing because it is massively SIMD (single instruction multiple data). A GPU has many threads that can simultaneously process the same instructions on multiple data very efficiently. The Lattice Boltzmann method performs the same operations on every lattice point.

Physically, the simulation visually appears to model correct fluid dynamics behavior. However, using some code checks, I have found that mass in the simulation is not conserved. This is a problem, since mass should always be conserved. This is likely a result of the equilibrium distribution function I am using and so I will need to check this in the future.

Issues have been found with the simulation when velocities exceed approximately $\frac{1}{3}$. Under these conditions, the fluid tends to stop being incompressible and compress. The low mach number expansion of the Maxwell-Boltzmann distribution also tends to deteriorate. In addition, the problem becomes ill-conditioned, yielding increased floating point error.

7.1 Performance

A widely used metric of Lattice Boltzmann codes is MLUPS or mega lattice updates per second, meaning how many millions of gridpoints can be updated in one second.

7.1.1 Single Threaded Performance

Core 2 X9650	3.00GHz	11.3 MLUPS
Xeon E5520	2.26GHz	8.8 MLUPS
Core 2 E8300	2.83GHz	10.6

Multi threaded performance scales almost linearly under shared memory systems using OpenMP. Using 4 threads on a Core 2 Quad X9650, 41.4 MLUPS are acheived.

7.1.2 Memory Bandwidth Constraints

The primary performance constraint for the current implementation of the LBM method is memory bandwidth. Relatively, on current CPUS, many more memory operations are performed than actual floating point calculations. CPUs have evolved much more quickly than their memory subsystems have.

This constraint is very observable when dealing with domain sizes near the size of the CPU cache. Caches are very fast memory on the CPU itself, but small in size typically in the range of several MB. Performance drops by more than a factor of 2 when the size of the domain grows from being smaller than the cache to being larger than the cache.

Taking this into account, performance can be improved by combining the streaming and collision steps into one loop as to not require two memory read and writes for every grid point.

7.1.3 Parallel Performance Scaling

Benchmarks are conducted on a cluster of dual processor Itanium 2 CPUs @1.6 GHz connected via SDR Infiniband.

7.2 Implementation in CUDA

As the primary performance constraint is memory bandwidth and the LBM algorithm is very SIMD (single instruction, multiple data), the architecture of GPUs is very attractive as an implementation platform.

Using many threads, over 629 MLUPS are realized, an order of magnitude faster than the fastest single CPU system.

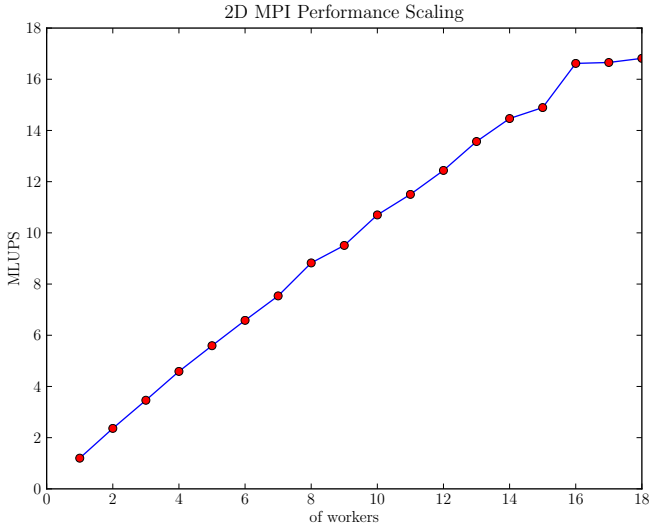
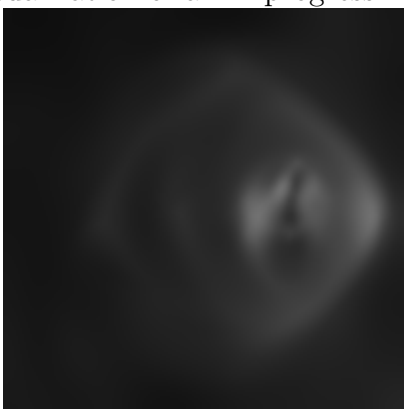


Figure 11: Visualization of an in progress fluid simulation



8 Conclusion

Lattice Boltzmann methods are a very attractive alternative to conventional fluid dynamics solvers since they exhibit accurate results and are much easier to parallelize. With the decline of Moore's law in serial performance, it has been realized in many-thread performance. Because of this, the Lattice Boltmzann method has become even more appealing as the future of fluid simulations.

Figure 12: Visualization of an in progress fluid simulation

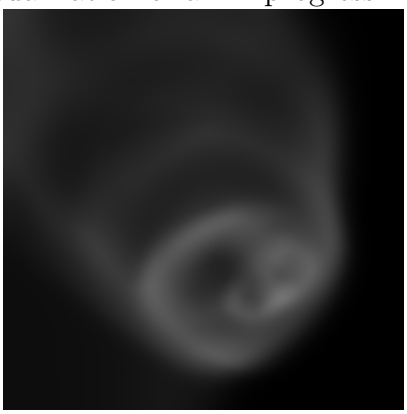
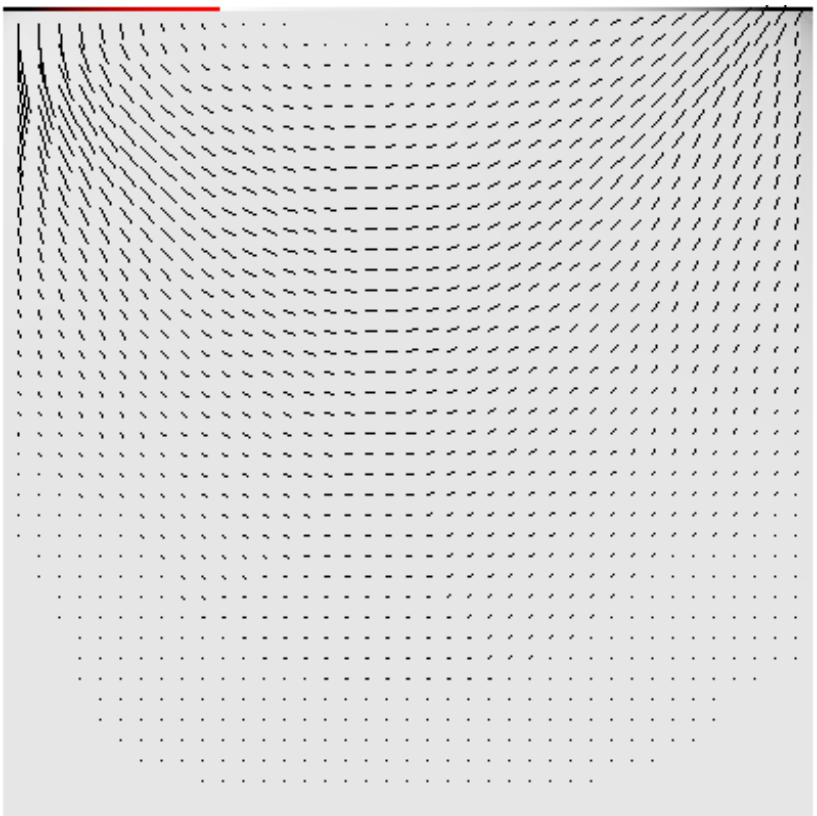


Figure 13: Visualization of an in progress fluid simulation using a velocity field



8.1 Applications

One application would be to simulate dispersion of smoke or bio toxins inside a large city such as New York. Emergency management teams could use this information in order to more effectively control such outbreaks and prevent against terrorist attacks.

8.2 Future Research

Improved Lattice Boltzmann Models can still be investigated such as the Multiple Relxation Time (MRT) model which performs collisions in moment space rather than distribution space, thereby improving numerical stability.

References

[1] File formats for vtk version 4.2. `http://www.vtk.org/VTK/img/file-formats.pdf`.

[2] 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.

[3] J. M. Buick and C. A. Greated. Gravity in a lattice boltzmann model. *Phys. Rev. E*, 61(5):5307–5320, May 2000.

[4] Xiaoyi He and Li-Shi Luo. Theory of the lattice boltzmann method: From the boltzmann equation to the lattice boltzmann equation. *Physical Review E*, pages 6811–6817, December 1997.

[5] Frdric Kuznik, Christian Obrecht, Gilles Rusaouen, and Jean-Jacques Roux. Lbm based flow simulation using gpu computing processor. *Computers & Mathematics with Applications*, 59(7):2380 – 2392, 2010. Mesoscopic Methods in Engineering and Science, International Conferences on Mesoscopic Methods in Engineering and Science.

[6] C. Körner N. Thürey, U. Rüde. Interactive free surface fluids with the lattice boltzmann method. 2005.

[7] U. Rüde N. Thürey. Free surface lattice-boltzmann fluid simulations with and without level sets. 2004.

[8] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, Oxford, 2001.