High Performance Computing with Python

Final Report

Kathapet Nawongs

5581839

tigerbee.nawongs@gmail.com

August 15, 2023

Contents

1

Introduction

The aim of this paper is to simulate liquid flow using the Lattice Boltzmann method (LBM). Some physics experiments are difficult to conduct because compressing the density is difficult in real life whereas LBM simulation is applicable to compressible liquid. The final experiment is to simulate liquid in a 2D cross-section container as the lid of the container slid off creating vortexes. The whole simulation can be executed in parallel and makes the simulation run a lot faster.

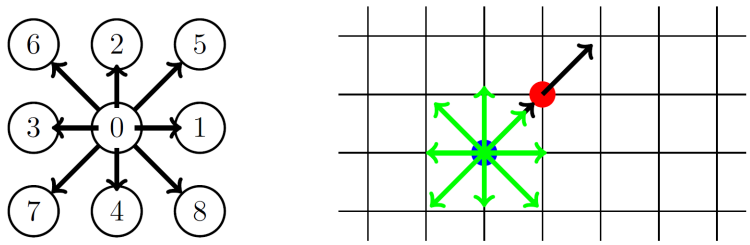
In Boltzmann, it considers the behaviour of particles as a whole which is a merge of the macroscopic and microscopic scale. In the microscopic scale, it considers the particles interaction and the macroscopic scale looks at the density, velocity and temperature averaged across all the particles. So LBM gets the advantage of handling complex geometries but don't have the noise obtained from fluctuations in microscopic level [1].

LBM is represented as a distribution function [2]. The distribution function is used to find particles in a certain part of the space. Since calculation (collision and streaming) in LBM is localized, LBM is adaptable for parallel computing [1]. An advantage of LBM to other methods like Conventional Navier Stokes is LBM's simplicity. In Navier Stokes, it is required to solve Poisson equation at each time step which is difficult to solve for its nonlinearity [1].

In Chapter 2 is the fundamental theory of Fluid Mechanics and LBM and parallelization. In Chapter 3 is mainly the implementation of the Lattice Boltzmann Equation, the streaming operator, boundary conditions and parallelization. In Chapter 4 is the results from Shear wave decay, Couette flow, Pouseille flow, Sliding Lid and parallelization. Chapter 5 is the conclusion.

2

Methods



(a) (b)

Figure 2.1: Descretized velocity in D2Q9 and uniform 2D grid

2.1 LBE & Streaming Operator

The Lattice Boltzmann scheme represents velocity and position space in descretized manner. It is descretized by dividing it into a 2D grid. The one used here is D2Q9 because the particle can move to either 8 directions from the initial position in Figure 2.1. For D2Q9, the velocity equation is Equation 2.1.

$$c = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{pmatrix}^T \tag{2.1}$$

The descritized Lattice Boltzmann equation is [1]:

$$f_i(\mathbf{x} + \mathbf{c}_i \cdot \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(x, t) \tag{2.2}$$

The particle f_i moves at velocity c_i to location $x + c_i \cdot \Delta t$ at timestep $t + \Delta t$ and a collision operator Ω_i. The streaming part is [1]:

$$f_i(\mathbf{x} + \mathbf{c}_i \cdot \Delta t, t + \Delta t) = f_i^*(\mathbf{x}, t) \tag{2.3}$$

which is the movement of particles in the vacuum.

2.2 Collision Operator

Ω_i in equation 2.2 can be replaced so that the whole equation can become [2]

$$f_i(\mathbf{x} + \mathbf{c}_i \cdot \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \omega \left(f_i(\mathbf{x}_j, t) - f_i^{\mathrm{eq}}(\mathbf{x}, t)\right) \tag{2.4}$$

The equilibrium population distribution function (PDF) is [2]

$$f_i^{eq} = \omega_i \rho \Big[1 + 3(\mathbf{c} \cdot \mathbf{u}) + \frac{9}{2}(\mathbf{c} \cdot \mathbf{u})^2 - \frac{3}{2}u^2\Big] \tag{2.5}$$

The density and velocity are required to be calculated in equation 2.5 [2]

$$\rho(\mathbf{x}_j, t) = \sum_i f_i(\mathbf{x}_j, t) \tag{2.6}$$

$$\mathbf{u}(\mathbf{x}_j, t) = \frac{1}{\rho(\mathbf{x}_j, t)} \sum_i \mathbf{c}_i f_i(\mathbf{x}_j, t) \tag{2.7}$$

The density is calculated by summing up the whole population and the velocity is calculated by summing up the population values multiplied with each channels

ω, which consists of the weights given for each channels, is defined for D2Q9 as [2]

$$\omega_i = \left(\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right) \tag{2.8}$$

The stationary channel is $\frac{4}{9}$, the straight channels are $\frac{1}{9}$ and the diagonal channels are $\frac{1}{36}$

2.3 Shear Wave Decay

In this experiment, two different distributions of the density and velocity are initialized at timestep $t = 0$. The first one is $\rho = \rho_0 + \varepsilon \sin(\frac{2\pi x}{L_x})$ and $u = 0$ and the second one is $\rho = 1$ and $u_x = \varepsilon \sin(\frac{2\pi y}{L_y})$. Both resulting in a sinusoidal profile and decaying over time because of the viscosity.

The results from both initialization is compared with the analytical prediction of kinematic viscosity. Its equation is [1]:

$$\nu = c^2\left(\frac{1}{\omega} - \frac{1}{2}\right) \tag{2.9}$$

with $c = \frac{1}{\sqrt{3}}$ which is the speed of sound.

The steps in shear waves are first calculating the equilibrium distribution function, then perform the streaming on the function, then perform the collision and returns to the initial step.

2.4 Couette Flow

Couette Flow is when fluid is in between two parallel plates, the top plate moves with a certain velocity in the x direction while the bottom plate is rigid [1].

The rigid wall's equation is when the particles are bounced back to the opposite channel because the momentum process is reversed when hitting the boundary back to the wet area[1]:

$$f_{\tilde{i}}(\mathbf{x}_b, t + \triangle t) = f_i^*(\mathbf{x}_b, t) \tag{2.10}$$

This is for conserving the momentum and mass at the boundary [2].

In the case of the top wall is

$$f_4(\mathbf{x}_b, t + \triangle t) = f_2^*(\mathbf{x}_b, t) \tag{2.11}$$
$$f_7(\mathbf{x}_b, t + \triangle t) = f_5^*(\mathbf{x}_b, t) \tag{2.12}$$
$$f_8(\mathbf{x}_b, t + \triangle t) = f_6^*(\mathbf{x}_b, t) \tag{2.13}$$

The moving wall equation is an extension of the rigid wall [1]:

$$f_{\tilde{i}}(\mathbf{x}_b, t + \triangle t) = f_i^*(\mathbf{x}_b, t) - 2w_i\rho_w\frac{\mathbf{c}_i \cdot \mathbf{u}_w}{c_s^2} \tag{2.14}$$

where u_w is the wall velocity, ρ_w is the extrapolated density.

It is to note that for the boundary conditions, the post-streamed PDF channels are assigned the pre-streamed PDF opposite channels' values.

The Couette flow's steps are first calculating the equilibrium distribution function, then performing the collision, then the streaming on the function and returns to the initial step.

2.5 Pouseille Flow

Pouseille Flow is also when fluid is in between two parallel plates. In this case both the top and bottom wall is rigid and there is a constant pressure gradient in the x direction from the inlet to the outlet. The inlet and outlet boundary condition is [1]

$$f_i^*(x_0, y, t) = f_i^{eq}(p_{in}, \mathbf{u}_N) + (f_i^*(x_N, y, t) - f_i^{eq}(x_N, y, t)) \tag{2.15}$$
$$f_i^*(x_{N+1}, y, t) = f_i^{eq}(p_{out}, \mathbf{u}_1) + (f_i^*(x_1, y, t) - f_i^{eq}(x_1, y, t)) \tag{2.16}$$

Equation 2.15 requires extra column nodes x_0 and x_{N+1}. x_0 is related to the column nodes of x_N and x_{N+1} is related to x_1. The pressure difference Δp in the experiment should be minuscule.

The Pouseille's steps are first calculating the equilibrium distribution function, then performing the collision, then performing boundary conditions with pressure variation, then the streaming on the function and returns to the initial step.

2.6 The Sliding Lid

In Sliding Lid, the north wall is moving horizontally, and the other three walls are rigid. Reynolds number is important in this experiment as the same number will give the same physics [1]. To recreate the standard flow in the box with vertices, the Reynolds number has to be 1000. The velocity will be kept small of around 0.1 and 0.2.

The Sliding lid's steps are first calculating the equilibrium distribution function, then performing the collision, then performing boundary conditions with moving wall, then the streaming on the function and returns to the initial step.

2.7 Parallelization

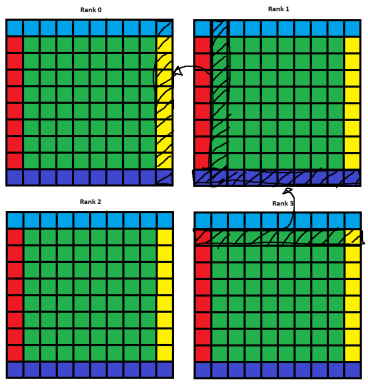


Figure 2.2: Parallelization of channel 6 from rank 3. The contiguous values from the non-ghost cells in rank 3 are sent to ghost nodes of rank 1 while the non-ghost cells from rank 1 are sent to the ghost cells of rank 0

The parallelization will be done using Message Passing Interface from python and domain decomposition. In Figure 2.2 is an example of the parallelization with 4 domains. Each of the domain consists of ghost nodes for receiving values from other domains and sending values to other domains. Each domain/rank performs its calculation for its subset and communicates with other domains. Each domain is handled by a processor. The number of processors are chosen that gives equal aspect ratios e.g. 1, 4, 9.

bwUniCluster, a parallel machine from Karlsruhe, provides access to processes used in section 4.5. There, simulations can be run that can not be handled in average PCs. When accessing the bwUniCluster, a certain amount of memory, tasks and nodes has to be allocated to perform a simulation. This is written as bash job file.

The Parallization Sliding Lid's steps are first calculating the equilibrium distribution function, then performing the collision, then the communication between the ranks, then the streaming on the function, then performing boundary conditions with moving wall and returns to the initial step.

3

Implementation

.

3.1 LBE & Streaming Operator

```python
import numpy as np

def calculate_eq_dist(rho_nm, u_anm, c_ai, w_i):
    u_anm_x_u_anm = np.einsum('anm, anm->nm', u_anm, u_anm)
    c_ai_x_u_anm = np.einsum('ai,anm->inm', c_ai, u_anm)
    cu_sq = c_ai_x_u_anm**2
    wrho_inm = np.einsum('i,nm->inm', w_i, rho_nm)
    f_eq = wrho_inm * (1 + 3 * c_ai_x_u_anm + 4.5 * cu_sq - 1.5
     * u_anm_x_u_anm)
    return f_eq
```

Listing 3.1: Equilibrium distribution function. Used `np.einsum()` for matrix multiplication between `u` `c` and ρ.

```python
import numpy as np

def streaming_operator(f_inm, c_ai):
    rho_nm = np.einsum('inm->nm', f_inm)
    for i in np.arange(1, 9):
        f_inm[i] = np.roll(f_inm[i], shift = c_ai.T[i], axis =
    (0, 1))
    return f_inm, rho_nm
```

Listing 3.2: Streaming operator in all 9 channels. `np.roll` shifts the probability density along the grid with a distance from `c_ai`.

3.2 Collision Operator

```python
import numpy as np

def collision(f_inm, c_ai, w_i, omega):
    # calculate density
    rho_nm = np.einsum('inm->nm', f_inm)
    # calculate average velocity
    u_anm = np.einsum('ia,inm->anm', c_ai.T, f_inm) / rho_nm
    f_eq = calculate_eq_dist(rho_nm, u_anm, c_ai, w_i)
    f_inm += ((f_eq - f_inm) * omega)
    return f_inm, u_anm, rho_nm
```

Listing 3.3: Calculates the collision by calculating density velocity and equilibrium then calculate the equilibrium PDF.

3.3 Couette Flow

```python
import numpy as np

def rigid_south_boundary(new_f_inm, f_inm):

    opposite = np.array([0, 3, 4, 1, 2, 7, 8, 5, 6])
    for i in np.array([4, 7, 8]):
        new_f_inm[opposite[i], -1, :] = f_inm[i, -1, :]
    return new_f_inm
```

Listing 3.4: Bounce back from rigid south boundary by storing the opposite channel values.

```python
import numpy as np

def moving_wall(new_f_inm, f_inm, u_anm, c_ai):
    opposite = np.array([0, 3, 4, 1, 2, 7, 8, 5, 6])
    u = [0, 1]
    rho_nm = np.einsum('inm->nm', f_inm)
    rho_mean = np.mean(rho_nm)
    sound_speed_squared = (1/np.sqrt(3))**2

    for i in np.array([2, 5, 6]):
        new_f_inm[opposite[i], 0, :] = f_inm[i, 0, :] - 2 * w_i
    [i] * rho_mean * (c_ai[:, i] @ u) / sound_speed_squared

    return new_f_inm
```

Listing 3.5: Bounce back from moving wall. Extension of rigid wall with wall velocity of 1. The matrix multiplication between `c_ai` and `u` results in a zero velocity for channel 4 negative velocity for channel 7 and positive velocity in channel 8.

3.4 Pouseille Flow

```python
import numpy as np

def pressure_variation(f_inm, rho_nm, u_anm, c_ai, w_i, d_inlet, d_outlet):
    # Inlet boundary condition
    f_eq_in = calculate_eq_dist_pressure(d_inlet * np.ones(nx), u_anm[:, :, -2], c_ai, w_i)
    f_neq_in = f_inm[:, :, -2] - calculate_eq_dist_pressure(rho_nm[:, -2], u_anm[:, :, -2], c_ai, w_i)
    for i in range(0, 9):
        f_inm[i, :, 0] = np.add(f_eq_in[i], f_neq_in[i])
    # Outlet boundary condition
    # equilibrium for outlet
    f_eq_out = calculate_eq_dist_pressure(d_outlet * np.ones(nx), u_anm[:, :, 1], c_ai, w_i)
    # non equilibrium for outlet
    f_neq_out = f_inm[:, :, 1] - calculate_eq_dist_pressure(rho_nm[:, 1], u_anm[:, :, 1], c_ai, w_i)
    for i in range(0, 9):
        f_inm[i, :, -1] = np.add(f_eq_out[i], f_neq_out[i])
    return f_inm
```

Listing 3.6: Periodic boundary conditions with pressure variation. Based on the equations in Section 2.5

3.5 Parallelization

```python
def Communicate(c, cartcomm, sd):
    sR, dR, sL, dL, sU, dU, sD, dD = sd

    # Send from left, receive on right
    sendbuf = c[:, :, -2].copy()
    recvbuf = c[:, :, 0].copy()
    cartcomm.Sendrecv(sendbuf=sendbuf,dest= dR, recvbuf=recvbuf, source = sR)
    c[:, :, 0] = recvbuf

    # Send from right, receive on left
    sendbuf = c[:, :, 1].copy()
    recvbuf = c[:, :, -1].copy()
    cartcomm.Sendrecv(sendbuf=sendbuf,dest=dL, recvbuf=recvbuf, source = sL)
    c[:, :, -1] = recvbuf

    # Send from bottom, receive on top
    sendbuf = c[:, 1, :].copy()
    recvbuf = c[:, -1, :].copy()
    cartcomm.Sendrecv(sendbuf=sendbuf, dest=dU, recvbuf=recvbuf, source = sU)
    c[:, -1, :] = recvbuf

    # Send from top, receive on bottom
```

```
23      sendbuf = c[:, -2, :].copy()
24      recvbuf = c[:, 0, :].copy()
25      cartcomm.Sendrecv(sendbuf=sendbuf, dest=dD, recvbuf=recvbuf
        , source = sD)
26      c[:, 0, :] = recvbuf
27
28      return c
```

Listing 3.7: Each of the rank communicates with one another. The `cartcomm.Sendrecv()` is used for sending the non-ghost cell values while receiving the values at the ghost cells. dR dL dU and dD are where the values from the current processor are being sent to and sR sL sU sD are where the values received came from. This function occurs before the streaming function to handle particles moving past the boundaries.
The parameter `c` is the equilibribium PDF - the array parallelized. The parameter `cartcomm` is the Cartesian communicator of the subdomains

4

Results

4.1 Shear Wave decay

In Shear wave decay, the density and velocity is supposed to decay to zero but it is not the case in 4.1 where it oscillates around 0.5 and in 4.2 it decays to half the amplitude by 14000 timesteps. The initial perturbation for both 4.1 and 4.2 is a sine wave.

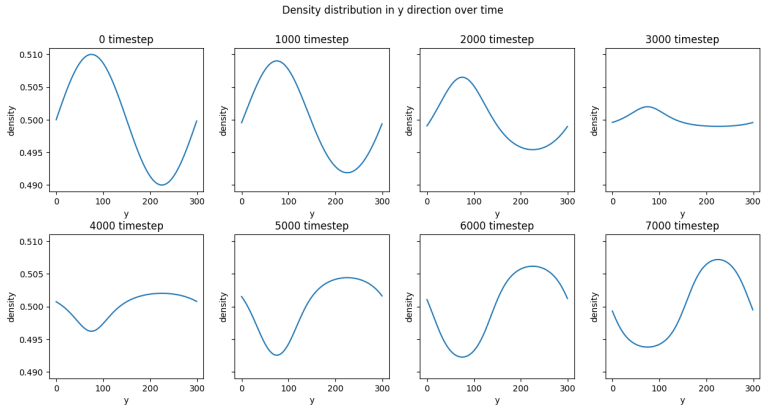


Figure 4.1: Density decay for epsilon of 0.01, omega of 1.3, ρ_0 of 0.5, 8000 timesteps and 300x300 grid. The density flips at around the 3000th timestep.

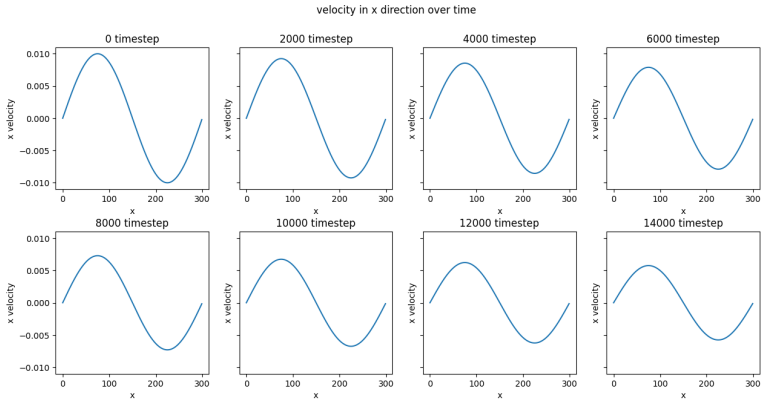
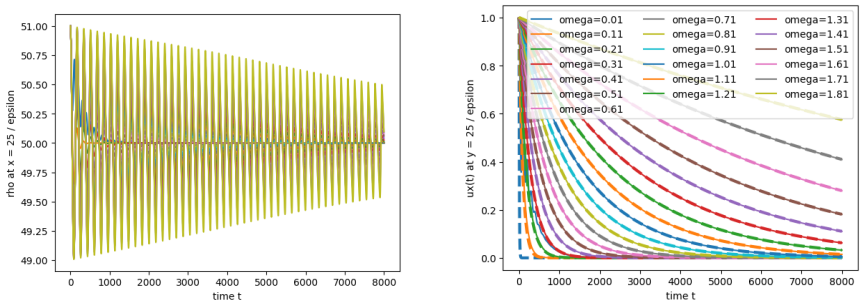


Figure 4.2: Velocity decay for epsilon of 0.01, omega of 1.3 and 16000 timesteps. The velocity in this case decays very slowly to a steady state. This performed in a 300-by-300 grid.



(a) Density amplitude decay

(b) Velocity amplitude decay

Figure 4.3: Decay for various omegas. These experiments used epsilon of 0.01, 8000 steps and 100-by-100 grid size. In figure 4.3a, the peak velocity decays but the overall velocity still oscillates, showing an over-relaxation. Figure 4.3b decays similarly to the theoretical perturbation. It shows for lower omegas, the velocity decays and reach convergence faster. This gives an under-relaxation.

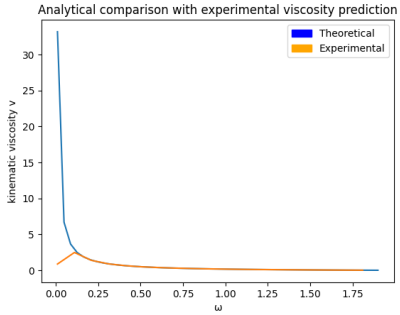


Figure 4.4: Kinematic viscosity for velocity perturbation with a grid of 100-by-100 compared with theoretical viscosity. The experimental viscosity follows very closely to the theoretical one for ω greater than 0.1 where it relaxes close to 0. Any viscosity for ω less than 0.1 is due to error in numpy calculation.

4.2 Couette flow

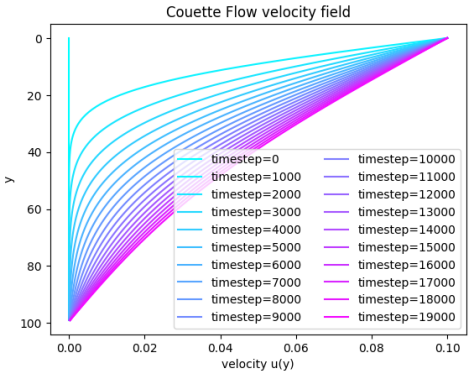


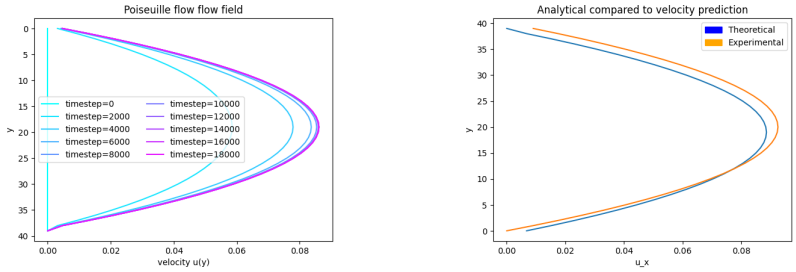
Figure 4.5: Evolution of velocity profile to 19000 timestep. Velocity in x-axis initialized to 0.1, grid of 100-by-100 and omega of 1.3. The velocity is initially large near the wall and later on propogates to the rest of the pipe until it is a diagonal line.

4.3 Poiseuille flow

In the Poiseuille flow experiment, the velocity profile is compared with the exact Poiseuille solution [1]:

$$u(y) = -\frac{1}{2\mu}\frac{\Delta p}{x_{out} - x_{in}} y(h - y)$$

where $\mu = \rho v$



(a) Evolution of velocity profile to 18000 timestep. The flow gives a parabolic graph at the steady state.

(b) Comparison of velocity profile at 20000 timestep to analytical. The prediction follows the analytical solution.

Figure 4.6: Density inlet is 1.005 and outlet is 0.995, grid of 40-by-80 and omega of 1.3.

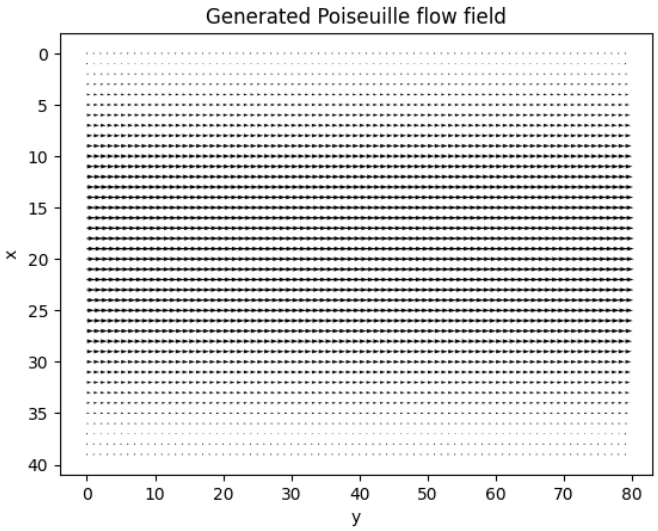
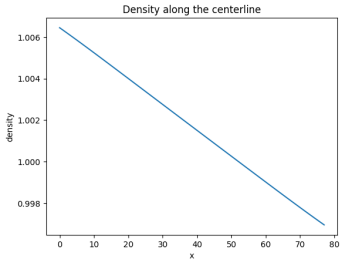
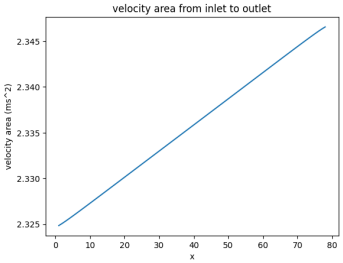


Figure 4.7: Flow field from poiseuille flow. Darker arrows represent higher magnitude velocities. The arrows go from left to right. This is used with quiver from matplotlib and the x and y velocity in the last timestep.



(a) The density along the centerline of the channel ρ (x,h/2). The density at the inlet is around 1.005, at the outlet is around 0.995. It is decreasingly linear from the inlet to outlet. This is due to fluid density being related to pressure via $p = c_s^2 p$[1]



(b) Area of the velocity profile of the channel. The increase in velocity from the inlet to the middle is by 0.95%. This slight increase is from the pressure differential between inlet and outlet

Figure 4.8: Poiseuille flow for last timestep

4.4 The Sliding Lid

The sliding lid requires a Reynolds number 1000 as a standard for flow patterns. The Reynolds number is calculated with uN/v [2] where u is the velocity, N is either the length or height of the grid and v is the viscosity. In fluid flow it is important for the Reynolds number to be similar to the geometrical aspects ratio.

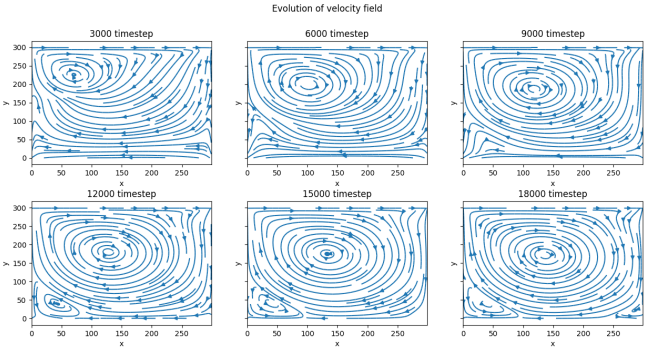


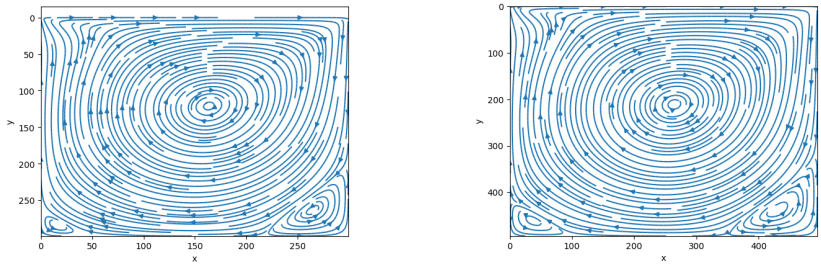
Figure 4.9: Evolution of the velocity flow field for Reynolds number 1000. Obtained Reynolds number 1000 by 300-by-300 grid, 1.3 omega and wall velocity 0.3. In the steady state, 2 eddies appear in the bottom.

4.5 Parallelization

The Lattice Boltzmann is scaled in the bwUniCluster where batch jobs are performed to allocate the number of CPU to the job. The `sbatch` submits

the job in the Linux system. In the bash file, it is required to specify the partition which in the experiment is `dev_multiple` because it does not require more than 160 processors.

To verify if the code works with scaling, 2 dimensions for many number of processors are tested on for their Million Lattice Updates per Second (MLUPS). The MLUPS is calculated by $\frac{L_x \times L_y \times nt}{t}$ where nt is the number of time steps and t is the processing time in seconds [1].



(a) Timestep of 100,000, grid of 300x300, omega of 1.3, wall velocity of 0.15. This is parallelized with 36 processors with `dev_multiple queue`.

(b) Timestep of 100,000, grid of 500x500, omega of 1.3, wall velocity of 0.15. This is parallelized with 81 processors with `dev_multiple queue`.

Figure 4.10: Comparison in parallelization with 300x300 and 500x500 grid. The flows are almost identical.

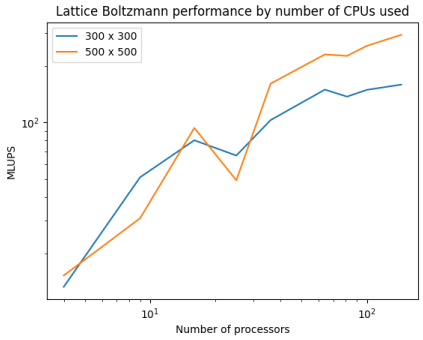


Figure 4.11: Measures Million Lattice Updates per Second (MLUPS) for 300-by-300 and 500-by-500 grid as a log-log scale. MLUPS diminishes as number of processors is greater than 100. MLUPS converges at higher value for larger grids.

5

Conclusion

This paper simulated Lattice Boltzmann in parallel and serial mainly with Python. The parallel part uses MPI to use multiple clusters. The multiple clusters are initally from PC in a small scale, then from BWUniCluster from Karlsruhe in a large scale. The parallelization method in the large clusters are performed using `sbatch`.

This paper performed on multiple experiments. It gave interesting results for Poiseuille flow, Couette flow and sliding lid. In Couette flow where there is a fixed and moving wall, the velocity steady state is a linear diagonal flow line. In Poiseuille flow the difference in pressure results in a quadratic velocity flow. The sliding lid gives a beautiful velocity field of vortex with multiple eddies. Since the equilibrium distribution and collision step is computationally heavy especially in large grids, it will take time for the velocity to reach the steady state. That is why parallelization is used to decompose the domain of the grid for reducing the runtime for sliding lid.

Bibliography

[1] Krüger Timm, H Kusumaatmaja, A Kuzmin, O Shardt, G Silva, and E Viggen. *The lattice Boltzmann method: principles and practice*. Springer: Berlin, Germany, 2016.

[2] AA Mohamad. *Lattice boltzmann method*, volume 70. Springer, 2011.