High-Performance Computing: Fluid Mechanics with Python

Mukesh Gudimallam (5168861)

March 27, 2024

universität freiburg

Contents

1 Introduction **3**

2 Methods **4**

2.1 Boltzmann Transport Equation 4
2.2 Lattice Boltzmann Method 4
2.3 Couette Flow 5
2.4 Poiseuille Flow 6
2.5 Sliding lid 6
2.6 Shear Wave Decay 7
2.7 Boundary Conditions 7
2.7.1 Rigid Wall 8
2.7.2 Moving wall 9
2.7.3 Periodic Boundary Condition with pressure gradient 9
2.8 Parallelization 10

3 Implementation **12**

3.1 Streaming 14
3.2 Collision 14
3.3 Bounce back condition 14
3.4 Couette Flow 15
3.5 Shear Wave 15
3.6 Poiseuille Flow 16
3.7 Sliding lid 17
3.8 MPI Sliding lid 18

4 Result **21**

4.1 Shear Wave decay 21
4.2 Couette Flow 24
4.3 Poiseuille Flow 25
4.4 Sliding Lid 25
4.5 Sliding Lid MPI 28

5 Conclusion **28**

1 Introduction

The recent developments in the field of High-Performance Computing (HPC) had a massive impact on the field of scientific research and engineering. A High-Performance Computing (HPC) system is typically a cluster that consists of tightly interconnected computers that work in parallel as a single system to perform large computational tasks [1]. Computational Fluid Dynamics (CFD) is one of the fields that reap benefits from the development of High-Performance Computing (HPC).

Computational Fluid Dynamics (CFD) is a numerical method of simulating steady and unsteady fluid motions using computation methods and hardware. It is a well-established method to replace experimental and analytical methods to aid the engineering design. They are used in a wide range of engineering applications such as Aerospace, Automotive, Chemical, etc., citecfd. The equations governing the fluid dynamics are difficult to solve analytically and therefore, numerical methods are employed to discretize the fluid dynamics. These methods demand significant computational resources, as the complexity of the system increases [3].

The Lattice Boltzmann Method (LBM) is a unique approach to fluid dynamics that operates at the mesoscopic level, unlike the Navier-Stokes equation which works at a macroscopic level. The Lattice Boltzmann Method (LBM) stands out for its numerical stability and flexibility for fluid simulations [4]. Originating from the Boltzmann Transport Equation (BTE) and grounded in kinetic theory, the Lattice Boltzmann Equation (LBM) simplifies the complex interactions of particles in a fluid through discretized time, space, and velocity sets [3]. This particle-based approach not only enhances computational efficiency but also simplifies parallel computing implementations, making it ideally suited for HPC environments [4]. In this report, we will go through the implementation of the Lattice Boltzmann Method (LBM) [1]in Python for different flow profiles.

[1]GitHub Repository for Code and simulation Implementation

2 Methods

The Lattice Boltzmann Method (LBM) offers an innovative perspective on fluid dynamics, diverging from traditional discretized partial differential equations. Instead, it adopts a statistical mechanics viewpoint, where fluids are represented by particle probability distributions across space and velocities. This approach begins with deriving the Navier-Stokes equations for incompressible fluids, transitioning to the Boltzmann transport equation, and culminating in the discretization of this equation through LBM. This method simplifies fluid simulation, allowing for efficient parallel computation, particularly on GPUs[11] [10] [12].

2.1 Boltzmann Transport Equation

The Boltzmann Transport Equation provides a foundation for understanding fluid dynamics from a microscopic perspective. It says that fluids can be modeled as systems of particles, each described by a probability density function f(x, v, t) indicating the likelihood of a particle's position and velocity at any given time [12].

The equation evolves by considering particle motion under forces, leading to a conservation equation without collisions [12]:

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} + \frac{\mathbf{F}}{m} \cdot \nabla_{\mathbf{v}}\right) f = 0 \tag{1}$$

A collision operator $\Omega(t)$ gives the interaction between particles.

$$(\frac{\partial}{\partial t} + v \cdot \nabla_x + F_m \cdot \nabla_v) f = \Omega(t) \tag{2}$$

We can derive macroscopic quantities of interest by integrating over f:
Density:

$$\rho(x,t) = \int f(x,v,t)\, dv \tag{3}$$

Momentum:

$$\rho(x,t)\mathbf{v} = \int v f(x,v,t)\, dv \tag{4}$$

2.2 Lattice Boltzmann Method

To simulate fluid dynamics, the Lattice Boltzmann Method (LBM) discretizes time, space, and velocity on a lattice. Unlike traditional methods, LBM uses a fixed set of velocity directions, such as nine directions in 2D simulations, including a stationary velocity. The method evolves the fluid's state by solving a discretized version of the Boltzmann transport equation, considering particle collisions through the Bhatnagar-Gross-Krook (BGK) approximation. This approximation assumes collisions drive the system towards an equilibrium distribution, defined by fluid properties and discretized velocities, facilitating the calculation of macroscopic fluid behavior[12].

In order to discretize, fix the velocity in Equation 2:

$$\frac{\partial f}{\partial t} + \mathbf{e}_i \cdot \nabla_x f + \mathbf{F}_m \cdot \nabla_v f = \Omega(t) \tag{5}$$

We also assume force F to be zero, under this assumption we get :

$$f_i(x + e_i \delta t, t + \delta t) - f_i(x,t) = \Omega_i(t) \tag{6}$$

Collision term from Equation 2 forces the distribution to slowly decay to equilibrium according to BGK [13].

Under the Bhatnagar-Gross-Krook (BGK) assumption, the collision term becomes:

$$\Omega_i = \frac{1}{\tau_1}(f_{\text{eq},i} - f_i) \tag{7}$$

Equilibrium Distribution under BGK assumption is given by [12]:

$$f_{\text{eq},i} = \omega_i \rho \left(1 + 3\frac{\mathbf{e}_i \cdot \mathbf{u}}{c} + \frac{9}{2}\left(\frac{\mathbf{e}_i \cdot \mathbf{u}}{c}\right)^2 - \frac{3}{2}\frac{|\mathbf{u}|^2}{c^2}\right) \tag{8}$$

To implement the LBM method , Equation 6 is split into two steps, Collision and Streaming.The left side of the equation is streaming, which handles the fluid movement and propagation. This step does not deal with collision therefore it can be done separately for each discrete velocities. The boundary conditions are dealt with in the streaming step.

After streaming the collision occurs, this is represented in the Right side of Equation 6. According to the assumption made in BGK, the collision relaxes to equilibrium. This is represented by [12]:

$$f_i = f_i - \frac{1}{\tau}(f_{\text{eq},i} - f_i) \tag{9}$$

The D2Q9 model as shown in Figure 1 is a common discretization scheme used in the Lattice Boltzmann Method for simulating fluid flows in two dimensions. It includes a grid with nine discrete velocities at each node: one stationary, four cardinal directions (East, West, North, South), and four diagonals. This model allows the simulation of fluid particles moving in two dimensions with these specified velocities, facilitating the calculation of macroscopic fluid properties like density and velocity while incorporating the effects of collisions and boundary interactions effectively.

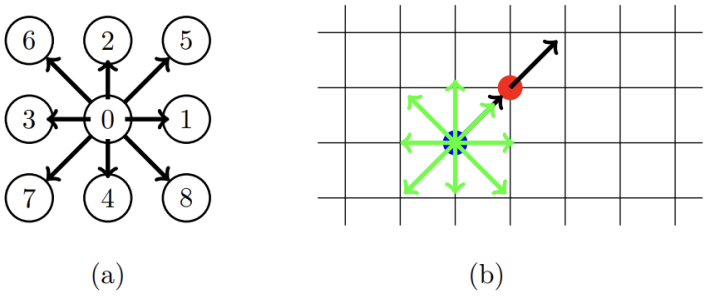

Figure 1: D2Q9 spatial representation of LBM[4]

2.3 Couette Flow

In this flow, a viscous fluid moves within the two parallel plates. Here one of the plates moves at a constant velocity while the other is fixed. This flow simulation demonstrates the linear velocity profile that is generated due to the motion in the top plate[9][11]

The generated profile can be represented by:

$$u(y) = \frac{U}{h}y \tag{10}$$

where $u(y)$ refers to the velocity of fluid at a distance y from the stationary wall. U denotes the velocity of the plate in motion, and h is the distance between the two plates. The generated velocity profile is due to the no-slip boundary condition at the surface of both plates and the shear stress in the whole fluid domain [11].

Shear stress τ of the liquid is defined as:

$$\tau = \mu \frac{\mathrm{d}u}{\mathrm{d}y} \tag{11}$$

This depicts that the shear stress that is exerted inside the fluid domain is directly proportional to the gradient of velocity across the layer of fluid. is a proportionality constant also known as dynamic viscosity.

2.4 Poiseuille Flow

Also referred to as Hagen-Poiseuille flow, depicts the laminar flow of a viscous fluid flowing through a pipe. The flow is distinctive with its parabolic velocity profile in the cross-section of the pipe, that is result of balance in force due to pressure and the drag generated due to viscosity. The equation that governs this flow is derived from the Navier-Stokes equation in the assumption of incompressible, steady, and laminar flow along with the no-slip boundary conditions at the walls. The velocity profile of the fluid is given by:

$$u(r) = \frac{\Delta P}{2L\mu} \left(R^2 - r^2\right) \tag{12}$$

where:

- μ refers to the dynamic viscosity of the fluid.
- R denotes the radius of the pipe.
- ΔP refers to the drop in pressure across the length of the pipe.
- $u(r)$ is the velocity of the fluid located at a distance r from the center of the pipe.

The maximum velocity in the fluid is located at the pipe center ($r = 0$) and represented in terms of pressure gradient ΔP, radius of the pipe R, and length L by:

$$u_{\max} = \frac{\Delta P R^2}{2L\mu} \tag{13}$$

The volumetric flow rate (Q) can be calculated as:

$$Q = \int_0^R 2\pi r u(r) dr = \frac{\pi \Delta P R^4}{8L\mu} \tag{14}$$

2.5 Sliding lid

Sliding Lid simulations are an integral benchmark for CFD analysis. Here, the behavior of a fluid in a 2D square cavity is explored when the top lid (boundary) is moved at a constant velocity and hence simulates a "Sliding Lid". This offers insights into vortex generation and complex flow patterns in viscous fluids. LBM is highly proficient in handling sliding lid due to its simple boundary conditions implementation [14].

The main focus is on the velocity and the vorticity ω that are important for understanding the flow dynamics. The Reynolds number (Re) can be described as $\frac{UL}{\nu}$, where U denotes the lid velocity, L refers to the length of the cavity, and ν represents the kinematic viscosity. Here $\omega = \nabla \times \mathbf{u}$ refers to vorticity.

The relaxation time can be calculated from Re with the following relation[14]:

$$\text{Rel time} = \frac{LU}{Re}$$

The vorticity can be explained by using the following relation that is derived from the curl of the Navier-Stokes equation:

$$\frac{\partial \omega}{\partial t} + \vec{u} \cdot \nabla \omega = \nu \nabla^2 \omega \tag{15}$$

2.6 Shear Wave Decay

Shear wave decay is an important simulation scenario for Computational Fluid Dynamics (CFD). Using the Lattice Boltzmann Method (LBM), sinusoidal velocity perturbations are introduced to the lattice. Attenuation of the velocity magnitude is observed due to fluid viscosity. This simulation plays a crucial role in learning the kinematic viscosity ν of the fluids [10].

When Implementing shear wave in LBM, first the lattice is initialized with uniform fluid density across the whole grid. Then, a sinusoidal velocity perturbation is introduced along a single dimension to generate a shear wave. The progression of waves with time gives information regarding the flid's viscosity. The amplitude of the wave decreases due to the resistance of the fluid to shear deformations [9] [11].

The attenuation of the wave can be formulated as:

$$u(x,t) = u_0 \exp(-\nu k^2 t) \sin(kx) \tag{16}$$

Where $u(x,t)$ denotes the velocity at location x at time t. u_0 refers to the initial amplitude of the perturbation. ν refers to the kinematic viscosity, which is calculated as $\frac{1}{3}\left(\frac{1}{w} - \frac{1}{2}\right)$, and k represents the wave number, which can be expressed as $k = \frac{2\pi}{L}$, where L is the length of the channel. This implies that waves in a shorter channel decay faster [10] [11].

The density ρ and velocity parameters of the fluid can be estimated by the following equations:

$$\rho = \sum_i f_i \tag{17}$$

$$u = \frac{1}{\rho} \sum_i f_i \vec{c}_i \tag{18}$$

2.7 Boundary Conditions

In the previous section, we discussed the fundamental equation (Equation) used for implementing the streaming and collision steps in the lattice Boltzmann method. In this section, we discuss the boundary conditions, addressing how particle bumps into boundaries. The boundary conditions discussed here are rigid walls, moving walls, and periodic boundary conditions (PBCs) with pressure gradients [7][8].

There are two types of boundary conditions in the lattice Boltzmann method: dry nodes and wet nodes. Dry nodes denote boundaries positioned between nodes, while wet nodes denote boundaries located on lattice nodes [7][8].

In this section, we primarily concentrate on dry nodes, as they are comparatively easier to implement than wet nodes. Handling boundary conditions introduces complexities, particularly when boundaries are situated on lattice nodes like wet nodes.

2.7.1 Rigid Wall

The rigid wall boundary condition in fluid dynamics is essential for accurately simulating the behavior of fluids near solid boundaries. The boundary condition ensures that the fluid adheres to the solid surface and that there is no relative velocity between the fluid and the wall, a condition known as the "no-slip" condition [6].

No slip condition is achieved in LBM through bounce-back boundary conditions. This is based on the principle that when the particle hits the wall at time t, it reflects in the opposite direction i at time $t + \Delta t$, this creates the no-slip or bounce-back condition [6].

The Equation for the bounce-back boundary condition in lattice-based simulations is given by:

$$f_i\left(\mathbf{x}_b, t + \Delta t\right) = f_{\text{opposite}(i)}\left(\mathbf{x}_b, t\right) \tag{19}$$

Where:

- $f_i(\mathbf{x}_b, t + \Delta t)$ represents the post-collision distribution function at the boundary node in the i-th direction at time $t + \Delta t$.
- $f_{\text{opposite}(i)}(\mathbf{x}_b, t)$ is the pre-streaming distribution function at the boundary node in the direction opposite to i at time t.
- w_i is the weighting factor for the i-th direction in the lattice model.
- ρ is the fluid density at the boundary node.
- $\mathbf{u}_w$ is the velocity of the moving wall.
- $\mathbf{c}_i$ is the lattice velocity vector in the i-th direction.
- c_s is the speed of sound in the lattice.

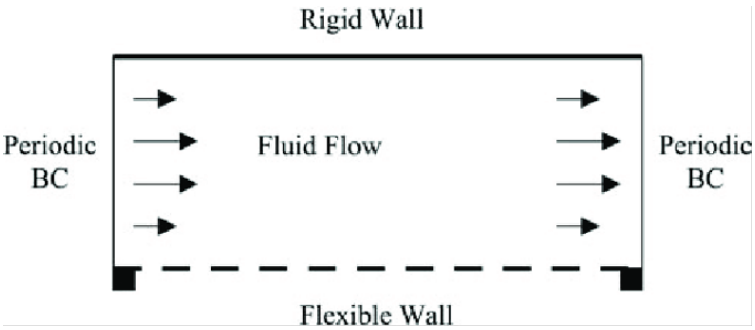

Figure 2: The Flow between rigid and flexible walls. The PBCs applied to the boundary at the left and right side of the flow field [7]

When dealing with stationary rigid walls (U_w= 0), the velocity of the fluid at the wall is assumed to be zero. This means that the particle distributions that stream to the wall node will simply reflect back without any alteration, preserving the mass and momentum [2]. In numerical simulations, the bounce-back rule is applied after the streaming step and before the collision step. The boundary condition is applied only at the boundary nodes x_b

2.7.2 Moving wall

The moving wall is similar to a rigid wall but complex, and the velocity at the boundary is non-zero. Here we incorporate the motion of the wall into the simulation. This involves interpolating between the no-slip condition and the moving wall condition. At the moving wall, the distribution functions are adjusted to account for the wall velocity U_w, allowing fluid particles to adhere to the wall's motion [8].

$$f_i\left(\mathbf{x}_b, t+\Delta t\right) = f_{\text{opposite}(i)}\left(\mathbf{x}_b, t\right) + 2w_i\rho\frac{\mathbf{u}_w \cdot \mathbf{c}_i}{c_s^2} \tag{20}$$

where:

- $f_i\left(\mathbf{x}_b, t+\Delta t\right)$ represents the post-collision distribution function at the boundary node in the i-th direction at time $t+\Delta t$.
- $f_{\text{opposite}(i)}\left(\mathbf{x}_b, t\right)$ is the pre-streaming distribution function at the boundary node in the direction opposite to i at time t.
- w_i is the weighting factor for the i-th direction in the lattice model.
- ρ is the fluid density at the boundary node.
- $\mathbf{u}_w$ is the velocity of the moving wall.
- $\mathbf{c}_i$ is the lattice velocity vector in the i-th direction.
- c_s is the speed of sound in the lattice.

2.7.3 Periodic Boundary Condition with pressure gradient

Periodic boundary conditions (PBCs) are utilized to simulate an infinite domain by replicating fluid behavior across the boundaries. In the Lattice Boltzmann Method (LBM), periodic boundaries are implemented by equating the distribution functions at opposite boundaries. To ensure an accurate simulation of fluid dynamics, these boundary conditions must be applied before the streaming step.

This approach is crucial for the proper implementation of the pressure gradient within the domain [6] [7].

The pressure and velocity conditions are given by Equation 21 and Equation 22 where the periodicity length is defined as:

$$L = N\Delta x,$$

where N represents the number of lattice nodes, and Δx is the lattice spacing. This definition allows for the simulation of continuous and seamless fluid movement across the computational domain, mimicking an effectively infinite space.

$$p(x,t) = p(x+L,t) + \Delta p \tag{21}$$

$$u(x,t) = u(x \pm L,t) \tag{22}$$

The pressure at any point x is the same as the pressure at a point one-period length L away, plus any change in pressure Δp that might occur over one period. The velocity at any point x is the same as the velocity at a point one-period length L away. The $\pm$ indicates that this applies in both the positive and negative directions along the axis [6] [8].

Using the ideal gas equation, the relationship between pressure and density can be expressed as $p = c_s^2\rho$, where ρ represents the fluid density and c_s denotes the speed of sound within the lattice units. This relationship allows the implementation of a density gradient to induce a flow, as pressure directly impacts the movement of fluid within the lattice.

To ensure specific pressure conditions, the pressure variation $\Delta p = p_{\text{in}} - p_{\text{out}}$ between the inlet and outlet is managed by updating the density values at these boundaries[8].

The boundary conditions for a fluid flow simulation are described using probability distribution functions (PDFs) and are given for the inlet and outlet nodes respectively:

For the inlet node x_0:

$$[f_i^*(x_0,t) = f_i^{\text{eq}}(\rho_{\text{in}}, \mathbf{u}) + (f_i^*(x_N,t) - f_i^{\text{eq}}(x_N,t))] \tag{23}$$

For the outlet node x_{N+1}:

$$[f_i^*(x_{N+1},t) = f_i^{\text{eq}}(\rho_{\text{out}}, \mathbf{u}) + (f_i^*(x_1,t) - f_i^{\text{eq}}(x_1,t))] \tag{24}$$

Here, f_i^* denotes the post-collision distribution function, and f_i^{eq} represents the equilibrium distribution function. By adjusting the distribution functions at the inlet and outlet nodes, the simulation maintains a density and pressure gradient across the computational domain to drive the fluid flow.

2.8 Parallelization

Parallel computing frameworks have played a significant role in the advancement of CFD simulations. For LBM, parallelization is very useful due to its inherent parallel structure and flexibility.

Using MPI we can achieve communication across the processors in distributed systems. In LBM simulations the computational domains are divided into subdomains therefore, MPI is important for implementing LBM simulation. LBM computational domains are partitioned into subdomains, each handled independently on separate processors. Effective inter-processor communication facilitated by MPI is essential to maintain the integrity and consistency of the simulation across domain boundaries, ensuring smooth coordination and synchronization of computational tasks [15].

Domain Decomposition- division of computational domain strategically into subdomains for a balance workload across processors. Target is to reduce the overlap and reduce the communication overhead for parallel efficiency.

Ghost Nodes-These are extra nodes that are added at the subdomain edges to facilitate data exchange between adjacent processors. This is to ensure accurate boundary condtions are maintained in the domain.

Processor Selection- Parallelization efficiency also depends on the careful selection of the count of processors where the communication to computation ratio is very crucial. Here, the processors are distributed in a matrix and the subdomains are divided among them [15].

$$S(\text{speedup}) = \frac{1}{(1-P) + \frac{P}{N}} \tag{25}$$

Ahmdal's law- This is used to find the maximum improvement to a system when a specific part is improved. This is used in MPI for finding insight for speedup in latency of executing a task for fixed workload. It can be expressed as:

3 Implementation

The D2Q9 model of the Lattice Boltzmann Method (LBM) mentioned in the 2.2, is implemented using Python. The implementation follows OOP's concepts where each flow profile is defined as an individual class and it inherits all the necessary common functions from its Base class. The UML Diagram for the overall code is depicted below in Figure 3.

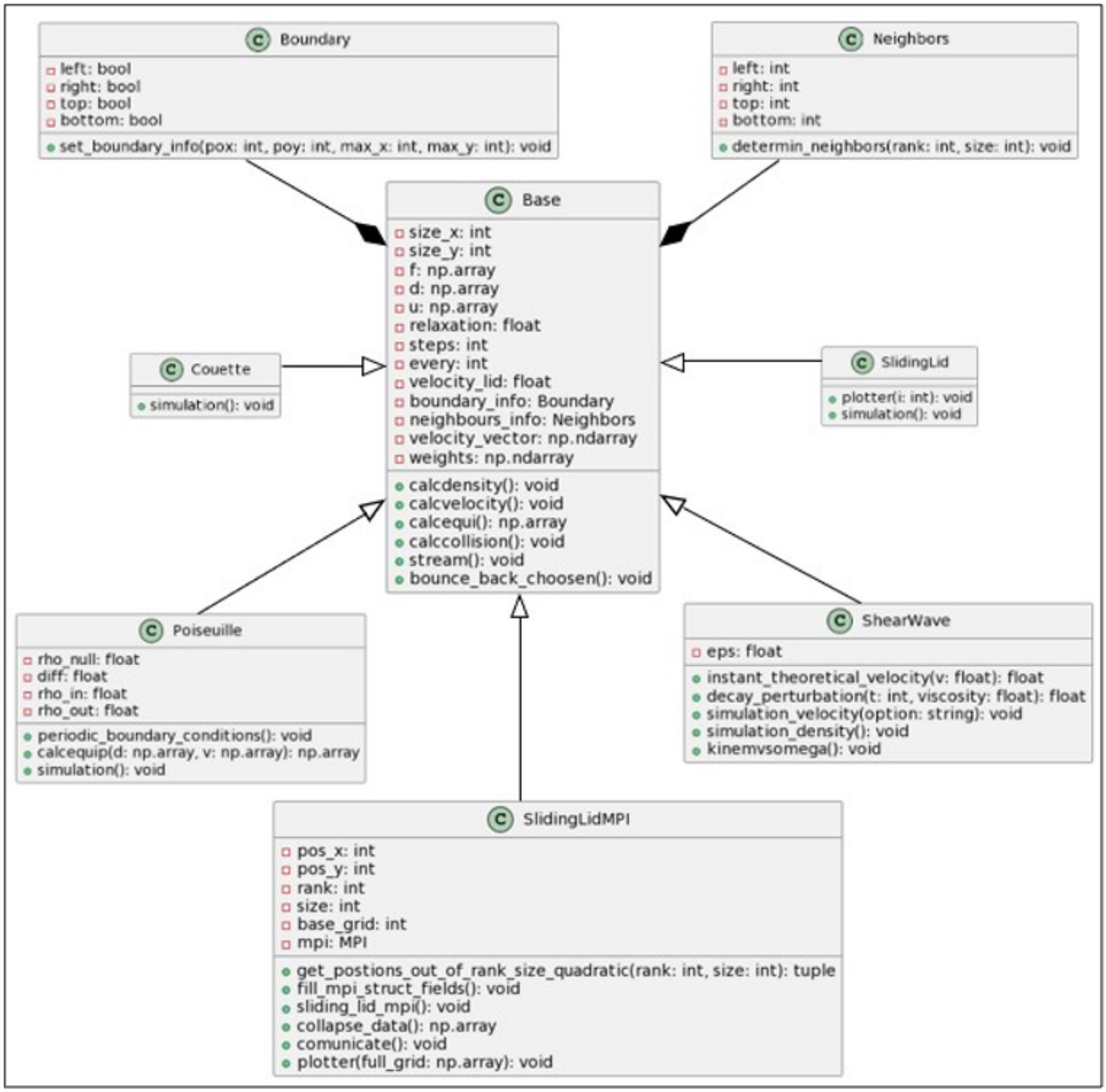

Figure 3: describes the UML Diagram of the entire code implemented [5]

Each flow profile is a derived class that inherits the common functionalities from the Base class. The Base class also has a composition relationship with the Boundary and the Neighbors class offering a better redundancy schema. All the classes mentioned in the above UML diagram use dataclass decorator therefore ensuring that there are no junk values stored in the data members upon start of the code as it provides an inbuilt default constructor while creating an object for the class. The basic input for the LBM code is a Lattice Grid having 'x' rows and 'y' columns. Each

Lattice point has 9 degrees of freedom as per the D2Q9 model mentioned in the 2.2. A Probability Density Function determines the occupancy of the particles in each Lattice point.

Base Class

The Base class implements the common functionalities such as streaming, collision, calculating equilibrium, and choosing the bounce-back conditions for the Lattice Grid. The Base class has the following Data Members and Member Functions

Figure 4: Base Class Structure

- **size_x** and **size_y** are integer variables that denote the size of the lattice.
- **f** denotes the probability density function in the format 'f(c, size_x, size_y)' numpy array where 'c' is the velocity vector according to the D2Q9 model.
- **d** denotes density in the format 'd(size_x, size_y)' numpy array.
- **u** denotes velocity in the format 'u(2, size_x, size_y)' where '2' denotes the x and y fields of the velocity vector.
- 'relaxation' denotes the relaxation constant (omega).
- 'steps' is the number of simulations to be performed for a fluid profile.
- 'every' is the intermediate update during simulation.
- 'velocity_lid' denotes the velocity of the moving wall if present.
- 'boundary_info' and 'neighbours_info' and objects of Boundary and Neighbors class respectively.
- 'velocity_vector' is a numpy array containing 9 pairs of values for 9 different directions according to the D2Q9 Model.
- 'weights' is a numpy array which is the weight matrix for calculating the collision term.

3.1 Streaming

In streaming, the PDF of all Lattice points in the Lattice Grid gets updated based on their velocity movement with respect to the D2Q9 Model. A special function in Python known as numpy.roll() is used to update all the Lattice points at once which results in very effective streaming operations.

```
def stream(self):
    for i in range(1, 9):
        self.f[i] = np.roll(
            self.f[i],
            self.velocity_vector[i],
            axis=(0, 1)
        )
```

Table 1: Base class member function to implement the streaming operation

3.2 Collision

To calculate the collision operation, the density and velocity values are calculated for the Lattice Grid as shown in 2.2. The equilibrium form is calculated using the imperial formula where the weight matrix gives the information about the chance of the particle to move in a specific velocity or direction. After which the updated Probability Density Function (PDF) after a collision for one-time step is calculated using the equilibrium form.

```
Algorithm:
def calcdensity(self):
    self.d = self.f.sum(axis=0)

def calcvelocity(self):
    self.u = (np.dot(self.f.T, self.velocity_vector).T / self.d)

def calcequi(self):
    return (((1 + 3*(np.dot(self.u.T, self.velocity_vector.T).T) +
              9/2*((np.dot(self.u.T, self.velocity_vector.T) ** 2).T) -
              3/2*(np.sum(self.u ** 2, axis=0))) * self.d).T * self.weights
                 ).T

def calccollision(self):
    self.f -= self.relaxation * (self.f - self.calcequi())
```

Table 2: Base class member functions implementing the collision operation for one time step

3.3 Bounce back condition

Boundary conditions for a fluid flow can be set dynamically using a set_boundary_info() method or the Boundary conditions can be initialized by sending the values to the data member while creating the Base class object. This is possible because the Boundary class in composition relationship with the Base class. After which the Boundary conditions are implemented as mentioned in the 2.7 using a bounce_back_chosen() method in the Base class.

```
Algorithm:
def bounce_back_choosen(self):
    if self.boundary_info.bottom:
        self.f[[2,5,6],:,1]=self.f[[4,7,8],:,0]
    if self.boundary_info.top:
        self.f[4, :, -2] = self.f[2, :, -1]
        self.f[7, :, -2] = self.f[5, :, -1] - 1 / 6 * self.velocity_lid
        self.f[8, :, -2] = self.f[6, :, -1] + 1 / 6 * self.velocity_lid
    if self.boundary_info.right:
        self.f[[3,6,7],-2,:]=self.f[[1,8,5],-1,:]
    if self.boundary_info.left:
        self.f[[1,5,8],1,:]=self.f[[3,7,6],0,:]

b = Base(size_x=100, size_y=50, steps=5000,
         boundary_info=Boundary(False, False, True, True))
# example where the boundary conditions are set for fixed top and bottom
   walls
```

Table 3: Base class member function to implement the Boundary conditions

3.4 Couette Flow

In Couette flow the fluid flows between a fixed wall and a moving wall as discussed. With the introduction of appropriate Boundary conditions while streaming, this results into the needed flow profile. Here, the Couette class uses all the common functions inherited from the base class to achieve the Flow profile 2.3.

```
from dataclasses import dataclass
import numpy as np

@dataclass
class couette(Base):
    def simulation(self):
        self.d = np.ones((self.size_x, self.size_y + 2))
        self.u = np.zeros((2, self.size_x, self.size_y + 2))
        self.f = self.calcequi()
        for i in range(self.steps):
            self.calcdensity()
            self.calcvelocity()
            self.calccollision()
            self.stream()
            self.bounce_back_choosen()

c = couette(size_x=100, size_y=50, steps=5000, every=10, velocity_lid=0.1,
    boundary_info=Boundary(False, False, True, True))
```

Table 4: Implementation of Couette flow for grid size 100x50 with lid velocity=0.1

3.5 Shear Wave

In computational physics, shear wave decay method is used to calculate the kinematic viscosity of a fluid. This is done by setting a sinusoidal velocity profile first and measuring how fast it decays. This

class consists implementation of shear wave decay for a sinusoidal velocity profile and a sinusoidal density profile as shown in 2.6. It also has one more method kinemvsomega() which gives us the relation between kinematic viscosity with respect to the change in omega value. Mathematical functions available from numpy library are used to perform the calculations more efficiently and faster on the entire Lattice Grid.

```
from dataclasses import dataclass
import numpy as np
from scipy.optimize import curve_fit

@dataclass
class shearwave(Base):
    eps: float = 0.05

    def instant_theoretical_velocity(self, v):
        return np.exp(-v * (2 * np.pi / self.size_y) ** 2)

    def decay_perturbation(self, t, viscosity):
        return self.eps * np.exp(-viscosity * (2 * np.pi / self.size_y) **
            2 * t)

    def simulation_velocity(self, option="None"):
        self.u[1, :, :] = self.eps * np.sin(2 * np.pi / self.size_y * np.
           arange(self.size_y)[:, np.newaxis])
        self.d = np.ones((self.size_y, self.size_x))
        self.f = self.calcequi()
        for step in range(self.steps):
            self.stream()
            self.calcdensity()
            self.calcvelocity()
            self.calccollision()
            kinematic_viscosity = 1 / 3 * (1 / self.relaxation - 1 / 2)

    def simulation_density(self):
        self.d = 1 + self.eps * np.sin(2 * np.pi / self.size_x * x)
        self.u = np.zeros((2, self.size_y, self.size_x), dtype=np.float32)
        self.f = self.calcequi()
        for step in range(self.steps):
            self.stream()
            self.calcdensity()
            self.calcvelocity()
            self.calccollision()

s = shearwave(size_x=150, size_y=200, steps=25000, every=3000, relaxation
   =1)
```

Table 5: Implementation of Shear wave decay for grid size 150x200 with relaxation constant=1

3.6 Poiseuille Flow

In this category of flow there exists a pressure difference between the two ends of the Lattice Grid due to which Periodic Boundary conditions mentioned in the 2.4 should be applied to get the resultant Flow profile. The member functions periodic_boundary_conditions() and calcequip() of

the poiseuille class help to achieve the necessary Periodic Boundary conditions, while the necessary normal Boundary conditions needed to realize the Flow profile are achieved through the member functions derived from the Base class.

```
from dataclasses import dataclass
import numpy as np

@dataclass
class poiseuille(Base):
    def periodic_boundary_conditions(self):
        self.calcdensity()
        self.calcvelocity()
        equilibrium = self.calcequip(self.d, self.u)
        equilibrium_in = self.calcequip(self.rho_in, self.u[:,-2,:])
        self.f[:, 0, :] = equilibrium_in + (self.f[:, -2, :] - equilibrium
            [:, -2, :])
        equilibrium_out = self.calcequip(self.rho_out, self.u[:, 1, :])
        self.f[:, -1, :] = equilibrium_out + (self.f[:, 1, :] -
            equilibrium[:, 1, :])

    def calcequip(self, d, v):
        return (((1 + 3*(np.dot(v.T, self.velocity_vector.T).T) +
                 9/2*((np.dot(v.T, self.velocity_vector.T) ** 2).T) -
                 3/2*(np.sum(v ** 2, axis=0))) * d).T * self.weights).T

    def simulation(self):
        shear_viscosity = (1/self.relaxation - 0.5) / 3
        self.d = np.ones((self.size_x + 2, self.size_y + 2))
        self.u = np.zeros((2, self.size_x + 2, self.size_y + 2))
        self.f = self.calcequi()
        for i in range(self.steps):
            self.periodic_boundary_conditions()
            self.stream()
            self.bounce_back_choosen()
            self.calcdensity()
            self.calcvelocity()
            self.calccollision()

p = poiseuille(size_x=100, size_y=50, steps=5000, every=10,
               velocity_lid=0.0, boundary_info=Boundary(False, False, True
                   , True))
```

Table 6: Implementation of Poiseuille flow for grid size 100x50 using periodic boundary conditions

3.7 Sliding lid

It is a closed box experiment in which it has a box filled with Fluid with bottom, right and left walls fixed and has a moving top lid, which induces the motion inside the Fluid as discussed in 2.5.Similar to Couette Flow the slidinglid class derives all the basic operations needed inherited from the Base class to realise the needed Flow profile.

```
from dataclasses import dataclass
import numpy as np

@dataclass
class slidinglid(Base):
    def simulation(self):
        self.d = np.ones((self.size_x + 2, self.size_y + 2))
        self.u = np.zeros((2, self.size_x + 2, self.size_y + 2))
        self.f = self.calcequi()
        for i in range(self.steps):
            self.stream()
            self.bounce_back_choosen()
            self.calcdensity()
            self.calcvelocity()
            self.calccollision()

re = 1000
base_length = 300
uw = 0.1
sl = slidinglid(size_x=base_length, size_y=base_length, every=100,
                steps=10000, relaxation=((2 * re) / (6 * base_length * uw
                    + re)),
                velocity_lid=uw, boundary_info=Boundary(True, True, True,
                    True))
```

Table 7: Implementation of Sliding Lid for grid size 300x300 with moving lid having velocity=0.1 and Reynolds number =1000

3.8 MPI Sliding lid

Implementing the Sliding Lid experiment in parallelisation using MPI library to fasten up the entire process using Amdahl's Law as mentioned in the 2.8.

To identify the neighbors of a Lattice point for establishing communication, enable necessary boundary conditions for the lattice and collapse the data at the end. The member functions determin_neighbors() from the Neighbor class object which exits in composition relationship with the Base class is used for the same as shown in Table 9.

```
from dataclasses import dataclass
import numpy as np
from mpi4py import MPI
@dataclass
class SlidingLidMPI(Base):
  def sliding_lid_mpi(self):
        for i in range(self.steps):
            self.stream()
            self.bounce_back_choosen()
            self.calcdensity()
            self.calcvelocity()
            self.calccollision()
            self.comunicate()
        full_grid = self.collapse_data()
   def comunicate(self):
        if not self.boundary_info.right:
            recvbuf = self.f[:, -1, :].copy()
            self.mpi.Sendrecv(self.f[:,-2, :].copy(), self.neighbours_info
                .right, recvbuf=recvbuf, sendtag = 10, recvtag = 20)
            self.f[:, -1, :] = recvbuf
        if not self.boundary_info.left:
            recvbuf = self.f[:, 0, :].copy()
            self.mpi.Sendrecv(self.f[:, 1, :].copy(), self.neighbours_info
                .left, recvbuf=recvbuf, sendtag = 20, recvtag = 10)
            self.f[:, 0, :] = recvbuf
        if not self.boundary_info.bottom:
            recvbuf = self.f[:,: ,0 ].copy()
            self.mpi.Sendrecv(self.f[:, :,1 ].copy(), self.neighbours_info
                .bottom, recvbuf=recvbuf, sendtag = 30, recvtag = 40)
            self.f[:, :, 0] = recvbuf
        if not self.boundary_info.top:
            recvbuf = self.f[:, :, -1].copy()
            self.mpi.Sendrecv(self.f[:, :, -2].copy(), self.
                neighbours_info.top, recvbuf=recvbuf, sendtag = 40, recvtag
                 = 30)
    re = 1000
base_length = 300
uw = 0.1
comm = MPI.COMM_WORLD
rank_1d = int(comm.Get_size() ** 0.5)
steps = 10000
slmpi = SlidingLidMPI(rank=comm.Get_rank(), size=comm.Get_size(),
                      size_x=base_length // rank_1d + 2,
                      size_y=base_length // rank_1d + 2,
                      relaxation=((2 * re) / (6 * base_length * uw + re))
                          ,
                      steps=steps, velocity_lid=uw, base_grid=base_length
                          , mpi=comm)
```

Table 8: Implementation of MPI Sliding Lid which gets its grid size, number of time steps, and the number of processors as the command line input from the user executing the mpirun command

```
import numpy as np

def get_postions_out_of_rank_size_quadratic(rank, size):
    return (rank % int(np.sqrt(size))), (rank // int(np.sqrt(size)))

def fill_mpi_struct_fields(self):
    self.pos_x, self.pos_y = self.get_postions_out_of_rank_size_quadratic(
        self.rank, self.size)
    self.boundary_info.set_boundary_info(self.pos_x, self.pos_y, int(np.
        sqrt(self.size)) - 1, int(np.sqrt(self.size)) - 1)
    self.neighbours_info.determin_neighbors(self.rank, self.size)
    self.d = np.ones((self.size_x, self.size_y))
    self.u = np.zeros((2, self.size_x, self.size_y))
    self.f = self.calcequi()
```

Table 9: Implementation of Neighbour Identification and setting appropriate Boundary conditions for each lattice in MPI Sliding Lid Implementation

4 Result

In this section, the Lattice Boltzmann Method (LBM) used for Computational Fluid Dynamics (CFD) is validated by realizing different Fluid profiles such as shear wave decay, Couette flow, Poiseuille flow, and finally sliding lid implementation. To validate the performance, the results are being compared to the analytical results in each flow profile. Finally, the Lattice Boltzmann Method is implemented in a parallel process to realize a fluid flow profile, showcasing the scalability and effectiveness of the Lattice Boltzmann Method.

4.1 Shear Wave decay

In shear wave decay, the velocity and the density decays are observed which are implemented using the algorithm stated in the 2.6. Also, the analytical value of the experiment is compared to its simulated value and finally, the change in the kinematic viscosity is observed with respect to the change of omega value. The density and velocity decay on the Lattice Grid of size 150x200 is implemented and the results are shown below.

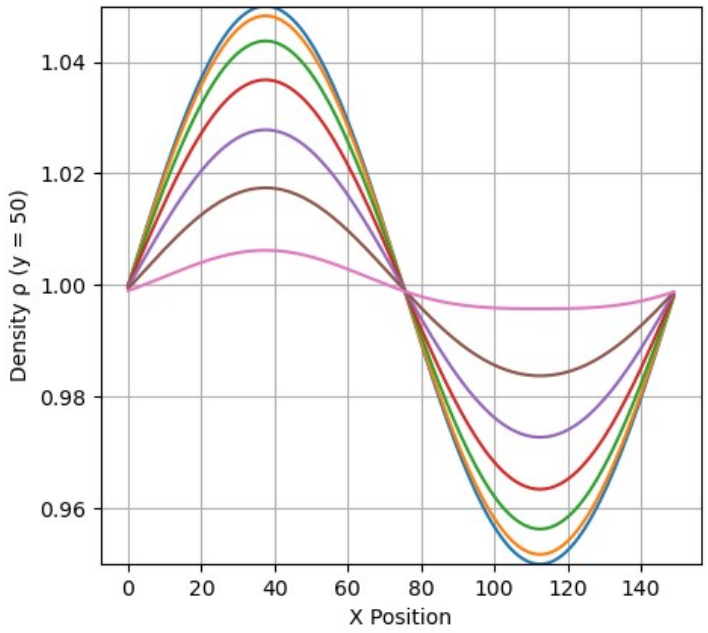

Figure 5: density decay over 70time steps for a Lattice Grid of size 150x200

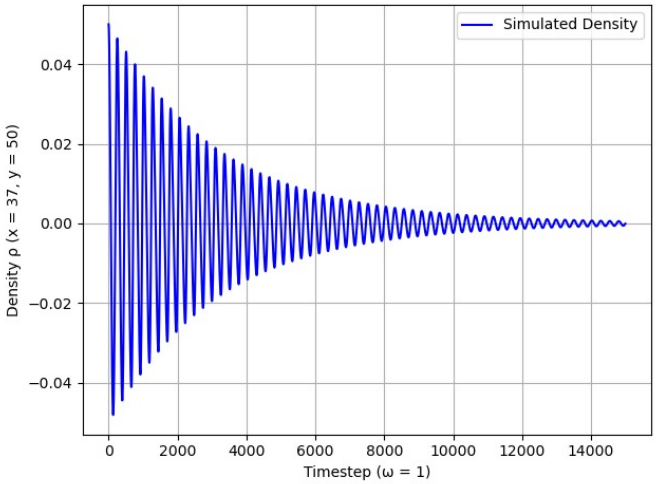

Figure 6: overall decay of the density function for 15000 time steps at rho(x=37,y=50) for a Lattice Grid of size 150x200

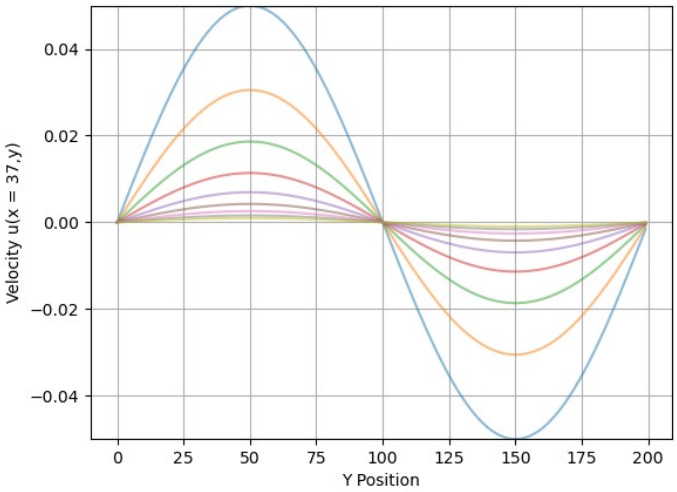

Figure 7: velocity decay in space for a Lattice Grid 150x200

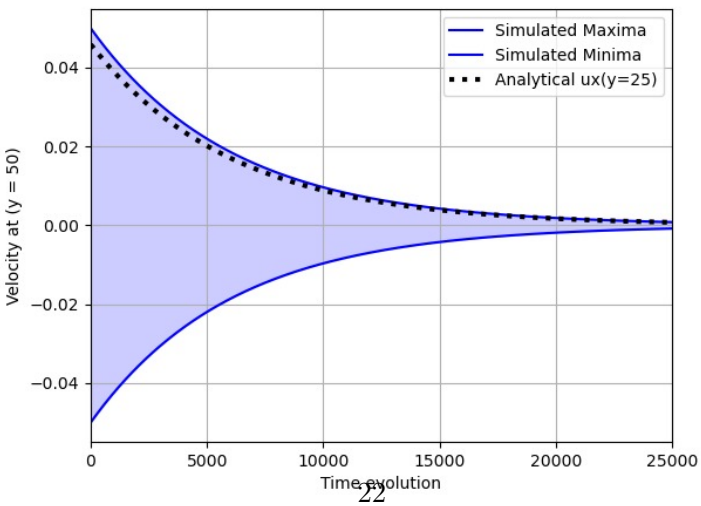

Figure 8: Velocity decay over time for 25000 time steps done on a Lattice Grid of size 150x200 and compared with its analytical value

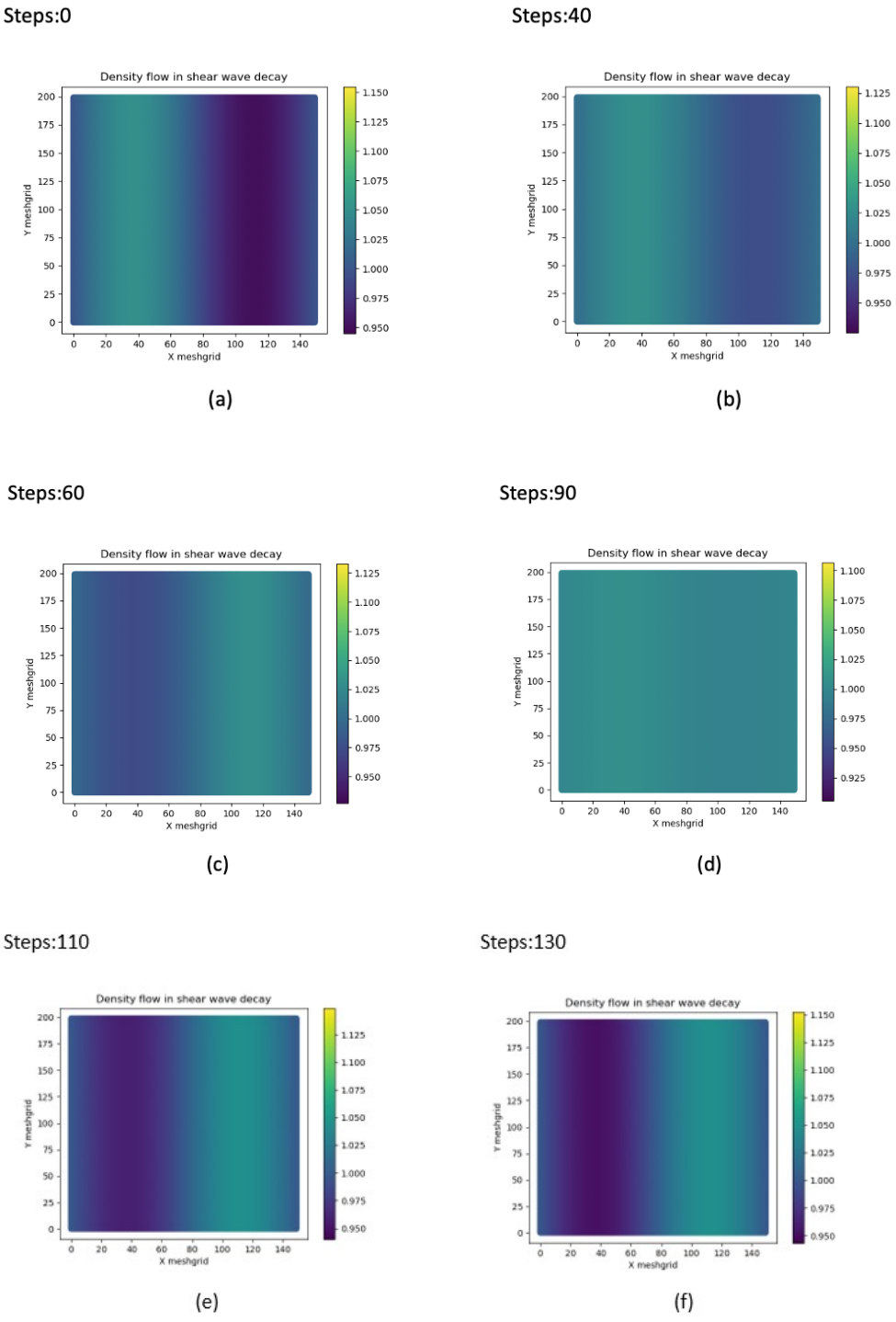

Figure 9: (a) – (f) shows the density flow across the Lattice Grid of size 150x200. We can observe that the density of Particles shifts from left side of the Lattice Grid to the right side of the Lattice Grid.

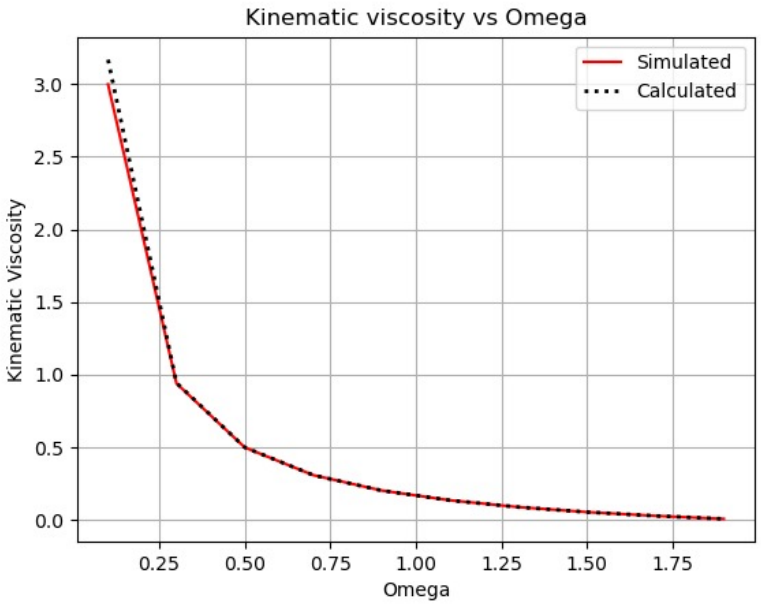

Figure 10: Kinematic viscosity decay over time for different values of omega. The simulated and analytical values obtained are calculated for the Lattice Grid of size 150x200 and for over 25000 time steps.

From the results obtained, the Lattice Boltzmann model (LBM) performs good as there is only negligible difference in the analytical and the simulated values obtained for shear wave decay.

4.2 Couette Flow

A Couette Flow profile is created for a Lattice Grid of size 100x50 using the algorithm mentioned in the 2.3 for 2000 time steps and compared with the analytical calculations.

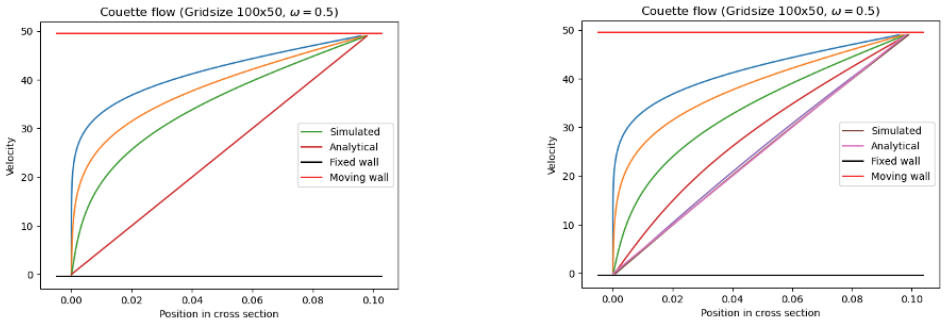

Figure 11: (a) is simulated for 350 time steps and (b) is simulated for 2000 time steps for the Lattice Grid 100x50 with lid velocity=0.1 and relaxation constant=0.5

From the results obtained the Couette flow reaches the predicted analytical value over a period of time.

4.3 Poiseuille Flow

Poiseuille Flow profile is created for a Lattice Grid of size 100x50 using the algorithm mentioned in the 2.4 for 2500 time steps and compared with the analytical calculations.

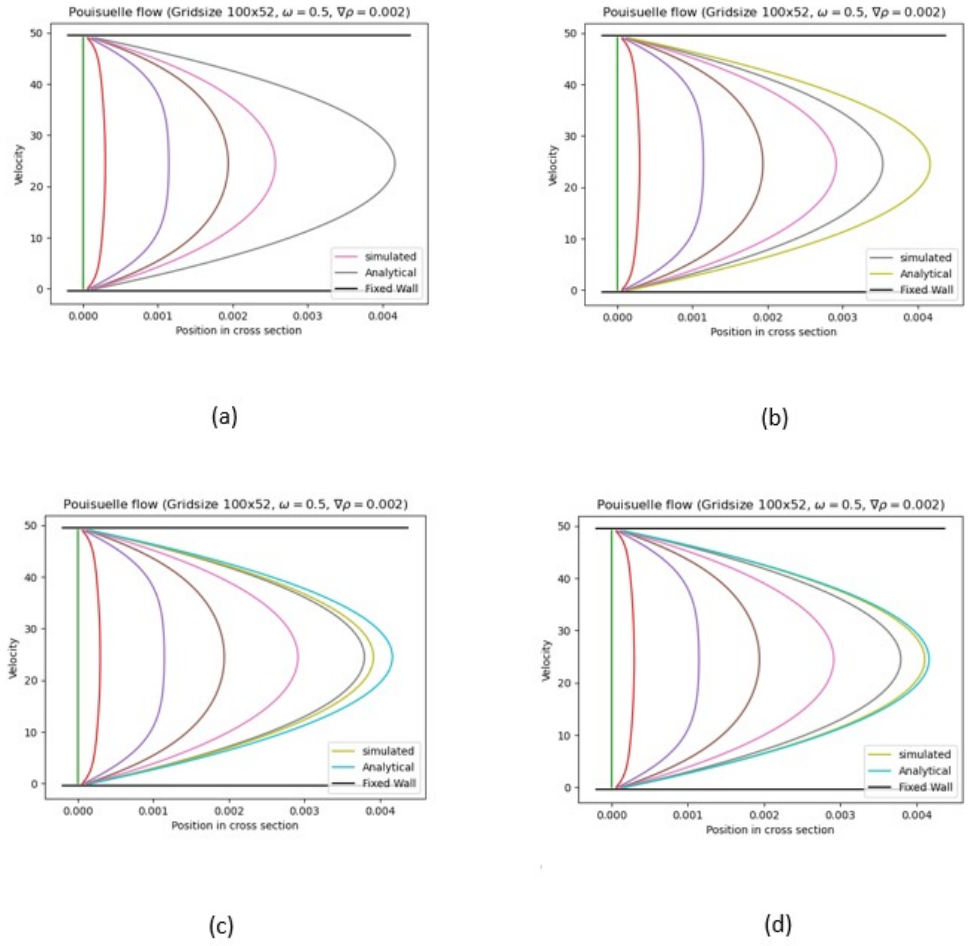

Figure 12: (a) is simulated for 500 time steps, (b) is simulated for 1000 time steps, (c) is simulated for 1500 time steps, and (d) is simulated for 2500 time steps for the Lattice Grid 100x50 with relaxation constant=0.5

From the results obtained the couette flow reaches the predicted analytical value over a period of time.

4.4 Sliding Lid

The sliding lid implementation is done using the algorithm given in the previous section. The output for a Lattice Grid of size 300x300 is observed having Reynolds number as 1000 executed over 10000-time steps. Then the relaxation constant is varied by varying the grid size as mentioned in the 2.5 and the output is observed.

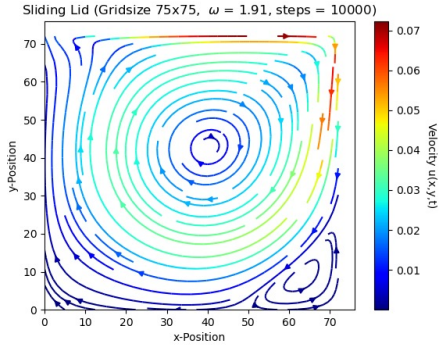

Figure 13: Sliding Lid implementation for grid size =75x75 with relaxation constant=1.91 simulated for 10000 time steps

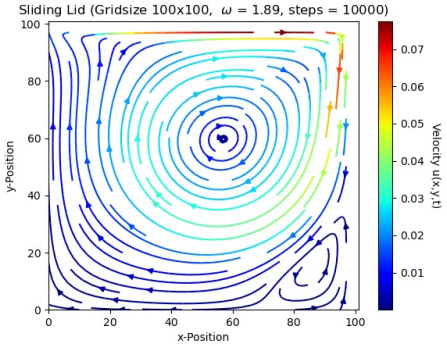

Figure 14: Sliding Lid implementation for grid size =100x100 with relaxation constant=1.89 simulated for 10000 time steps

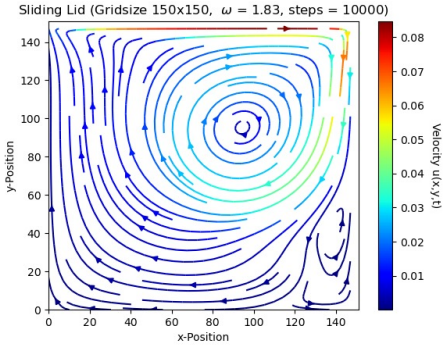

Figure 15: Sliding Lid implementation for grid size =150x150 with relaxation constant=1.83 simulated for 10000 time steps

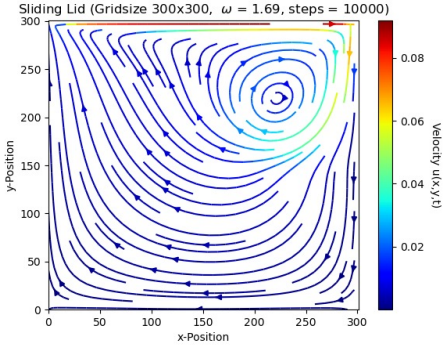

Figure 16: Sliding Lid implementation for grid size =300x300 with relaxation constant=1.69 simulated for 10000 time steps

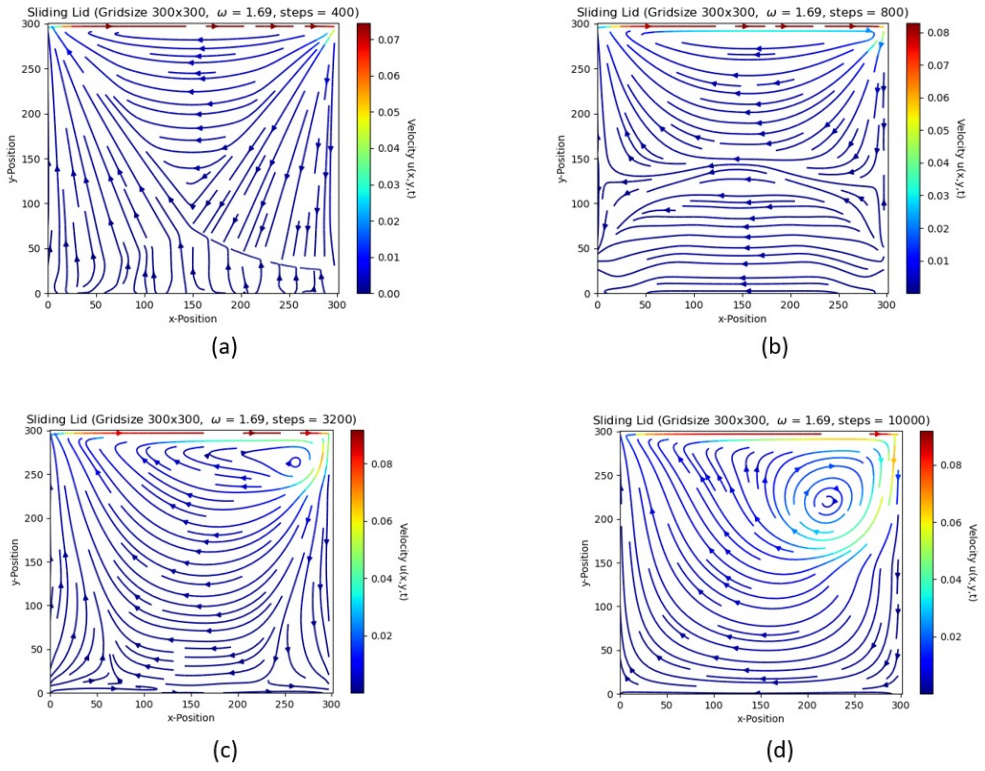

Figure 17: (a)-(d) denoted the progression of sliding lid implementation for Lattice grid size 300x300 with relaxation constant=1.69

4.5 Sliding Lid MPI

The parallel version of the sliding lid is implemented using the algorithm mentioned in the 2.8 and the behavior of time taken for the task to complete with respect to scaling is observed.

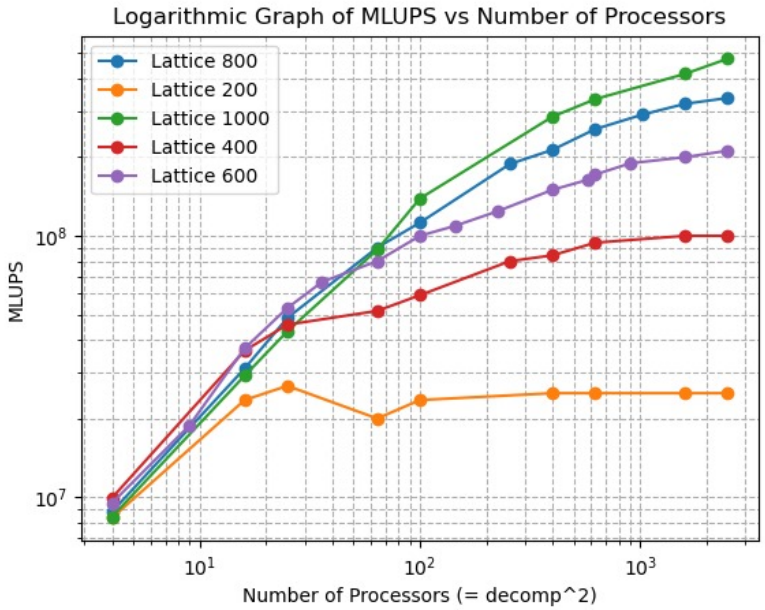

Figure 18: Sliding Lid MPI implementation for different grid sizes giving us MLUps vs Number of Processors

Here we observe that as the number of processors increases the time taken for the task decreases following Amdahl's speed-up law. But after some point, the communication overheard between the processors degrades the performance, at this point the increase in the number of processors would result in a meager performance increase or even decrease in performance. This can be observed on the Lattice Grid 200 and Lattice grid 400 very clearly in which after a certain increase in the Number of processors, the performance tends to decrease.

5 Conclusion

In this report, we explored the Lattice Boltzmann Method (LBM) for Computational Fluid Dynamic Simulations emphasizing its theoretical and practical implementation. Lattice Boltzmann Method's scalability for parallel computing showcases its high potential in working with High-Performance Computing Clusters. Using the theoretical concepts of LBM we have implemented multiple complex fluid simulations to validate the LBM method by comparing the simulated and the analytical values of the simulations. The evidence suggests that parallelization of the task significantly enhances the performance of the system following Amdahl's law and parallelized LBM implementation using Python and MPI reinforces LBM's stability, robustness, and suitability for complex Computational Fluid Dynamic simulations. Furthermore, the implementation in Python makes it more robust, supported, and efficient in performing mathematical operations across the Lattice Grid using special Python libraries.

References

[1] *NVIDIA, "High Performance Computing Glossary," Available:* https://www.nvidia.com/en-us/glossary/high-performance-computing/ *, [Accessed January 10, 2024]*

[2] *Siemens, "CFD Simulation," Available:* https://www.plm.automation.siemens.com/global/en/our-story/glossary/cfd-simulation/67873 *, [Accessed January 10, 2024].*

[3] *Jiaxing Qi, " Effiecient Lattice Boltzman Simulation on large scale HPC systems", Available:*https://publications.rwth-aachen.de/record/688936/files/688936.pdf?subformat=pdfa

[4] *Lars Pastewka and Andreas Greiner. "Hpc with python: An mpi-parallel implementation of the lattice boltzmann method"*

[5] *PlantUML, "A PlantUML Example," Available: Plantumlexample*

[6] *Clarkson University, "Boundary Conditions in Fluid Mechanics,"* [Online]. Accessed on [27-02-2024]. Available: https://web2.clarkson.edu/projects/subramanian/ch560/notes/Boundary%20Conditions%20in%20Fluid%20Mechanics.pdf

[7] *Flow Between Rigid and Flexible wall*, [Online]. Available: https://www.researchgate.net/figure/Flow-between-rigid-and-flexible-walls-The-periodic-boundary-condition-was-applied fig1_241700075. [Accessed: 27-02-2024].

[8] *Krüger Timm et al. The lattice Boltzmann method: principles and practice. Springer: Berlin, Germany, 2016.*

[9] *Zhen Chen, Chang Shu, and Danielle S. Tan, "Immersed boundary-simplified lattice Boltzmann method for incompressible viscous flows," Physics of Fluids, vol. 30, p. 53601, 2018.*

[10] *S. Succi, "The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond," Oxford University Press, 2001.*

[11] *S. Chen and G. D. Doolen,"Lattice Boltzmann Method for Fluid Flows," in Annual Review of Fluid Mechanics, vol. 30, pp. 329-364, 1998.*

[12] *"Lattice Boltzmann methods [Online]. Available:* https://andrew.gibiansky.com/blog/physics/lattice-boltzmann-method/ *. [Accessed: 03-02-2024].*

[13] *P. L. Bhatnagar, E. P. Gross, and M. Krook., "A Model for Collision Processes in Gases Phys. Rev. 94 (3 May 1954)*

[14] *P. Lallemand and L.-S. Luo, "Theory and Practice of the Lattice Boltzmann Method: A Tool for Computational Fluid Dynamics," Progress in Energy and Combustion Science, vol. 28, no. 4, pp. 249-283, 2002.*

[15] *C. Zhu, J. Liu, L. Feng, and X. Deng, "Research on the Simulation of PF-LBM Model Based on MPI+CUDA Mixed Granularity Parallel," AIP Advances, vol. 8, p. 65017, 2018.*