High Performance Computing: Fluid Mechanics with Python

Pooja Sheregar (5453456)

March 28, 2024

Contents

1 Introduction

Fluid dynamics is a fundamental area of physics and engineering that explores how gases and liquids behave. This field is crucial for numerous applications, from designing more efficient vehicles and predicting weather changes to understanding natural phenomena like river currents and blood flow. However, the traditional approach in fluid dynamics involves solving complex mathematical equations, notably the Navier-Stokes equations. These equations describe the motion of fluid substances but are challenging to solve analytically, especially for turbulent or complex flow patterns [1].

In order to overcome this challenge, the Lattice Boltzmann Method (LBM) has emerged as a powerful alternative for simulating fluid flows. Unlike traditional approaches that focus on solving the Navier-Stokes equations directly, LBM simplifies the problem by modeling fluid as a collection of particles on the lattice. LBM describes the behavior of fluids at the microscopic level. This particle-based approach makes it well-suited for numerical simulations. LBM's unique approach allows it to efficiently tackle problems that are difficult to solve with traditional methods, including flows involving complex boundaries, turbulent patterns, and multi-phase phenomena[2].

The objectives of this report are:

1. Exploring the fundamentals of the Lattice Boltzmann Method (LBM) and understanding its mathematical and operational principles.
2. Discretizing the fluid into a lattice domain (LBE) and implementing it.
3. Demonstrating the validation through simulation of shear wave decay, Couette flow, and Poiseuille flow.
4. Evaluating the performance and scalability of LBM in high-performance computing environments.

The implementation is done in Python using NumPy library and message passing interface (MPI) with mpi4py for parallelization. LBM offers simplified computational models for complex flows, particularly in dealing with boundary conditions and interfaces between different phases. Moreover, its inherent parallelizability makes it well-suited for large-scale simulations on modern high-performance computing platforms. By leveraging local interactions within its lattice framework, LBM facilitates efficient, scalable, and accurate fluid dynamics simulations.

The code implementation and related files are available in GIT.[1]

[1]GitHub Repository

2 Methods

The Boltzmann Transport Equation (BTE) and the Lattice Boltzmann Method (LBM) are fundamental to understanding the behavior of particles within a fluid and simulating fluid dynamics, respectively. Both play important roles in computational fluid dynamics (CFD) they provide the microscopic interactions of particles and the macroscopic flow properties observed in fluids. In this section, we explore the Boltzmann transport equation and the process of discretizing BTE to obtain the Lattice Boltzmann Equation [2].

2.1 Boltzmann Transport Equation

BTE is a statistical equation that describes the behaviors of the particle velocities in a system involving temperature gradients or densities such as fluids or gases. It shows how the distribution $f(\vec{x}, \vec{v}, t)$ changes with time due to the movement of particles called streaming and interaction between particles called Collison [2, 3]. The probability distribution function of particle velocities in a fluid in the absence of external force is given by BTE:

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla f = \Omega(f) \tag{1}$$

In the presence of external force, the equation can be written as follows [2, 3]:

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla f + \vec{F} \cdot \nabla_{\vec{v}} f = \left(\frac{\partial f}{\partial t}\right)_{\text{collision}} \tag{2}$$

Where:

- f is the distribution function
- $\Omega(f)$ is the collision operator, which describes the change in f due to particle collisions
- $\frac{\partial f}{\partial t}$ is the time derivative of the distribution function f
- $\vec{v} \cdot \nabla f$ represents the convective derivative, describing how particles move through space
- $\vec{F} \cdot \nabla_{\vec{v}} f$ accounts for the acceleration of particles due to external forces $\vec{F}$
- $\left(\frac{\partial f}{\partial t}\right)_{\text{collision}}$ is the collision term, representing the change in f due to particle interactions

The BTE representing the rate of change in the distribution $f(\vec{x}, \vec{v}, t)$ that changes with time is given by the below equation by substituting in Equation 2 (as given in the lecture) :

$$\frac{\partial f(\vec{x}, \vec{v}, t)}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f(\vec{x}, \vec{v}, t) + \vec{a} \cdot \nabla_{\vec{v}} f(\vec{x}, \vec{v}, t) = C(f(\vec{x}, \vec{v}, t)). \tag{3}$$

Where:

- $\frac{\partial f}{\partial t}$ is the partial derivative of the distribution function $f(\vec{x}, \vec{v}, t)$ with respect to time t, indicating the change in distribution with time.
- $\nabla_{\vec{x}} f$ is the gradient of the distribution function with respect to position $\vec{x}$, representing the spatial variation of the distribution.

[2]Unless cited otherwise, Some of the contents used in this report are inspired by the lecture material of Prof. Andreas Greiner on High-Performance Computing: Fluid Mechanics with Python at University of Freiburg as presented at Summer Semester of 2023 .

- $\vec{v} \cdot \nabla_{\vec{x}} f$ describes how particles carry their properties as they move through space at a velocity $\vec{v}$.
- $\vec{a} \cdot \nabla_{\vec{v}} f$ is the force term, where $\vec{a}$ is the acceleration the particles experience due to external forces, causing a change in their velocity distribution.
- $C(f)$ is the collision operator, capturing the effect of particle interactions.

The collision term from Equation 2 is approximated using the Bhatnagar-Gross-Krook (BGK) model, by simplifying it to a relaxation towards equilibrium it can be written as [9]:

$$\left(\frac{\partial f}{\partial t}\right)_{\text{collision}} = -\frac{1}{\tau}(f - f^{eq}) \tag{4}$$

$\frac{1}{\tau}$ is the inverse of relaxation term τ which the rate at which the distribution function f approaches its equilibrium state f^{eq}. $f - f^{eq}$ represents the deviation of the distribution function f from the equilibrium distribution f^{eq}.

We assume f relaxes towards equilibrium f^{eq} due to collision between particles with constant relaxation time τ according to BGK [[9]:

$$\frac{d}{dt} f(\vec{r}, \vec{v}, t) = -\frac{f(\vec{r}, \vec{v}, t) - f^{eq}(\vec{v}, \rho, \vec{u}, T)}{\tau} \tag{5}$$

The negative sign indicates that this process reduces the difference between the current state and equilibrium, thereby driving the system towards equilibrium.

2.2 Lattice Boltzmann Method

The Lattice Boltzmann Equation (LBE) is a numerical approach for simulating fluid flow by discretizing the Boltzmann equation. It considers the probability of particle positions and velocities within the lattice, allowing the efficient modeling of fluid dynamics in various scenarios. LBM discretizes the BTE along with the streaming and collision process which provides a unique and efficient framework for simulating fluid dynamics. It offers significant advantages over traditional CFD methods, in terms of computational simplicity and adaptability to handle complex geometries and boundary conditions [4].

The discretization involves dividing the physical space into a lattice and velocity space into a set of microscopic velocity vectors. The fundamental equation of LBM is the discretized version of BTE which is continuous and is given by [5]:

$$\frac{df(\vec{x}, \vec{v}, t)}{dt} = -\frac{f(\vec{x}, \vec{v}, t) - f^{eq}(\vec{v}; \rho(\vec{x}, t), \vec{u}(\vec{x}, t), T(\vec{x}, t))}{\tau} \tag{6}$$

Where:

- f^{eq} is the statistical equilibrium distribution function
- $T(\vec{x}, t)$ is the temperature at position $\vec{x}$ and time step t.
- τ is a relaxation time.
- $\rho(\vec{x}, t)$ is the macroscopic density at position $\vec{x}$ and time t.
- $\vec{u}(\vec{x}, t)$ is the macroscopic velocity at position $\vec{x}$ and time t.

Relaxation time τ determines how quickly the fluid converges to equilibrium. if the τ is high, the convergence to equilibrium is slow.

The main aspects of discretization are: space, time, and particle velocities [5, 7]

1. **Spatial Discretization:** The simulation domain is divided into lattice nodes, representing voids and solids, forming a structured mesh.
2. **Temporal Discretization:** Time is divided into discrete steps, with the system's state updated at each step.
3. **Velocity Discretization:** Particle velocities are restricted to a finite set, typically modeled by D2Q9 for 2D simulations and D3Q19 or D3Q27 for 3D simulations.

The spatial domain is discretized into an equidistant 2D grid lattice as illustrated in Figure 1. For the velocity space and time, the discretization is chosen such that $c_i \Delta t = \Delta x_i$, ensuring a consistent movement of particles between lattice nodes in one time step [5].

- c_i – distance between interpolation points in velocity space.
- Δx_i – distance between points on the spatial lattice.
- Δt – time step.

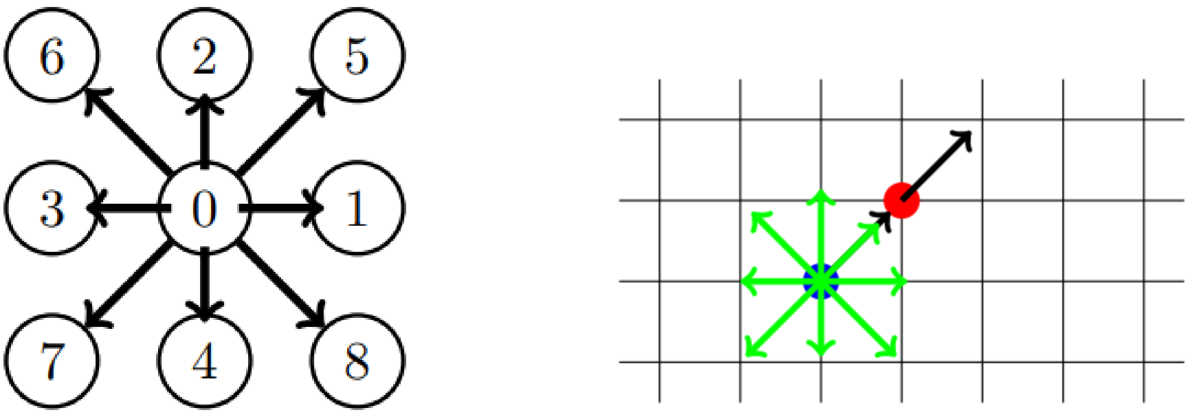


Figure 1: Discretization of the BTE. Left-velocity space discretization into nine directions. Right - two-dimensional lattice for the spatial discretization.

In the D2Q9 model, every lattice node is linked to its eight closest neighbors, along with a stationary state, permitting nine potential directions of particle movement, including remaining at the current node. Direction 0 signifies the population of molecules at rest. This captures fluid flow in two dimensions. Matrix c represents the velocity set of nine possible discrete velocities that particles can have in a two-dimensional lattice.

$$\mathbf{c} = \begin{pmatrix} 0 & 0 & -1 & 0 & 1 & -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \end{pmatrix}^{\mathrm{T}}$$

The discrete BGK Boltzmann equation can be written as [8] :

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

$$\underbrace{\text{streaming}}_{\text{process}} \qquad\qquad \underbrace{\text{collision}}_{\text{process}}$$

where:

- $f_i(\mathbf{x}, t)$ is the particle distribution function, which represents the probability of finding a particle with velocity $\mathbf{c}_i$ at position $\mathbf{x}$ and time t.
- $\mathbf{c}_i$ is the discrete velocity set in the direction i.
- Δt is the discrete time step.
- ω is the relaxation parameter, which determines the rate at which the distribution function relaxes towards equilibrium.
$$\omega = \frac{\Delta t}{\tau}$$
- $f_i^{\text{eq}}(\mathbf{x}, t)$ is the equilibrium distribution function, calculated based on the macroscopic properties of the fluid, such as density and velocity.

Fundamental operations within LBM include streaming and collision processes, which update particle distributions and handle interactions between particles, respectively. Boundary conditions are applied to simulate physical boundaries, and parallelization strategies are employed for computational efficiency.

Equilibrium distribution function f_i^{eq} in nine direction [8]:

$$f_i^{\text{eq}}(\mathbf{x}_j, t) = w_i \rho(\mathbf{x}_j, t) \left(1 + 3\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x}_j, t) + \frac{9}{2}[\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x}_j, t)]^2 - \frac{3}{2}\mathbf{u}^2(\mathbf{x}_j, t)\right)$$

where:

- $\mathbf{w}$ is a set of weights associated with the respective discrete velocities in the D2Q9 model.
- $\mathbf{c}_i$ are the discrete velocity vectors in the lattice.
- $\rho(\mathbf{x}_j, t)$ is the macroscopic density at position $\mathbf{x}_j$ and time t.
- $\mathbf{u}(\mathbf{x}_j, t)$ is the macroscopic velocity of the fluid at position $\mathbf{x}_j$ and time t.

The weights vector $\mathbf{w}$ is defined as:

$$\mathbf{w} = \left(\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right)$$

After particles stream across the lattice (moving from one lattice node to another), the collision step relaxes the distribution functions towards this equilibrium state. This relaxation process models the effects of microscopic collisions and interactions, allowing the simulation to capture macroscopic fluid behavior like flow and viscosity.

3 Implementation

In this section, we will discuss the step-by-step implementation of LBM and the parallelization technique used to enhance computation efficiency. The fundamental equation of LBM is translated into Python code to simulate fluid dynamics, leveraging the D2Q9 lattice discretization.

3.1 Streaming

To perform the streaming step in LBM the domain is initialized with a grid size and a set of velocity directions. The lattice Boltzmann D2Q9 model is used, represented by the $\vec{v}$ and c array containing velocity vectors in 2D space. w array contains associated weights for each direction.

The streaming function executes the core operation of the streaming step as shown in Listing 1. It loops over all velocity directions and shifts the distribution functions f across the lattice according to the velocity vectors. This is achieved using *np.roll*, which moves values in the array by a specified number of places along the given axis, simulating particle movement.

```python
import numpy as np

# Define lattice velocity vectors for the D2Q9 model
c = np.array([[0, 1, 0, -1, 0, 1, -1, -1, 1],
              [0, 0, 1, 0, -1, 1, 1, -1, -1]]).T

# Perform the streaming step, shifting distribution functions across the
    lattice
def streaming(f):
    for i in range(9):
        f[i,:,:] = np.roll(f[i,:,:], c[i], axis=(0, 1))
    return f
```

Listing 1: Python code for the streaming step of LBM

3.2 Collision

The collision step in the Lattice Boltzmann Method (LBM) plays a crucial role as it models the interaction between particles within the lattice. This guides the system towards an equilibrium state based on local density and velocity.

Collision steps work on the principle that velocities at each node will gradually relax towards equilibrium at constant relaxation time. This behavior is described by BGK [9] as shown in Equation 4 and Equation 5 and is implemented in Listing 2.

The collision function handles particle interactions within the lattice, while the equilibrium function calculates the equilibrium state based on the density d and velocity v at each lattice node. Subsequently, the collision function adjusts the distribution function to converge towards equilibrium.

```
import numpy as np
# Define weights for each of the velocity directions in D2Q9
w = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])
#Perform the collision step, adjusting the distribution function towards
    the equilibrium
def collision(f,v,d,relaxation=0.5):
    eq = equilibrium(d,v)
    f -= relaxation * (f-eq)
    return f
#Compute equilibrium distribution function based on density and velocity
def equilibrium(d,v):
    vel_mag = v[0,:,:] ** 2 + v[1,:,:] ** 2
    temp = np.dot(v.T,c.T).T
    sq_velocity = temp ** 2
    f_eq = (((1 + 3*(temp) + 9/2*(sq_velocity) - 3/2*(vel_mag)) * d ).T *
        w).T
    return f_eq
```

Listing 2: Python code for the collision and equilibrium step of LBM

3.3 Couette Flow

Couette flow refers to the viscous flow in the space between two parallel plates, one rigid and the other moving at a constant velocity U_w as shown in Figure 2. Any change in the fluid's velocity across the gap between the plates is only to the viscous drag because the flow is shear-driven with no pressure gradient. It examines the development of the velocity profile across the fluid due to the movement of the top plate, applying no-slip boundary conditions at the walls. The validation can be done by implementing a moving wall.

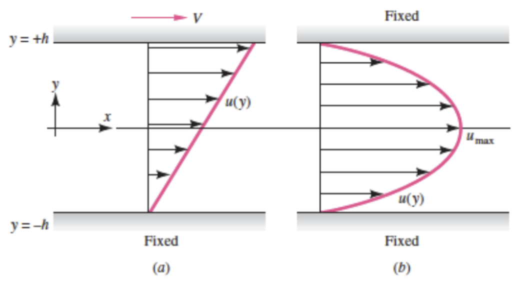


Figure 2: Illustration of flow between parallel plates (a) Couette flow: No pressure gradient, upper plate moving (b) Poiseuille flow: pressure gradient, both plates fixed [10]

The analytical solution for velocity profile $u_x(y)$ in Coutte flow is given by [10] :

$$u_x(y) = \frac{Y - y}{Y} U_w \tag{7}$$

where:

- $u_x(y)$ is the horizontal velocity of the fluid at a vertical position y.
- Y is the distance between the two parallel plates.

- y is the vertical position at which the velocity is measured from the bottom stationary plate.
- U_w is the velocity of the top plate, which moves in the horizontal direction.

The implementation involves a stationary bottom wall where the velocity is set to zero. The top wall, which is moving at velocity U_w uses a bounce-back boundary condition. The array c holds the velocity vectors for the D2Q9 model. Each vector corresponds to one of the nine possible directions particles can travel in the LBM grid. There are four cardinal direction vectors, four diagonal vectors, and one zero vector (representing particles that stay at the same node). The variable lid_vel represents the speed of the top plate in Couette flow as shown in Listing 3. The den array is initialized to ones, representing a uniform density across the lattice. The vel array is initialized to zeros, indicating that the fluid is initially at rest before the top plate begins moving.

```python
import numpy as np

lid_vel=0.1
# Initial conditions for density and velocity
den=np.ones((x,y+2))
vel=np.zeros((2,x,y+2))

den=np.ones((x,y+2))
vel=np.zeros((2,x,y+2))
f=equilibrium(den,vel)
for step in range(steps):
    den=density(f)
    vel=velcoity(f,den)
    f=collision(f,vel,den)
    f=streaming(f)
    f=boundry_condition(f,lid_vel)
#rest of the code
# boundary conditions for couette
def boundry_condition(f,lid_vel):
    f[2, :, 1] = f[4, :, 0]
    f[5, :, 1] = f[7, :, 0]
    f[6, :, 1] = f[8, :, 0]
    f[4, :, -2] = f[2, :, -1]
    f[7, :, -2] = f[5, :, -1] - 1 / 6 * lid_vel
    f[8, :, -2] = f[6, :, -1] + 1 / 6 * lid_vel
    return f
#rest of the code
Analytical_vel = lid_vel*1/50*np.arange(0,50)
```

Listing 3: Python code for the Coutte flow implementation

3.4 Poiseuille Flow

The Poiseuille flow is the type of flow in which fluid flow is driven by a pressure gradient between two parallel plates(non-moving walls) as shown in Figure 2. The flow is due to the constant pressure between the inlet and outlet. The resultant flow exhibits a parabolic velocity profile due to the viscous drag within the fluid, which is greater at the walls and minimal at the center. This effect creates a velocity gradient perpendicular to the flow direction [10].

The analytical solution for velocity is given by [10]:

$$u_x(y) = -\frac{1}{2\rho\nu}\frac{dp}{dx}y(Y-y) \tag{8}$$

where:

- $u_x(y)$ is the horizontal velocity component of the fluid at a vertical position y within the channel.
- $\frac{dp}{dx}$ is the pressure gradient along the direction of flow.
- ρ is the fluid density.
- ν is the kinematic viscosity of the fluid.
- Y is the total distance between the two parallel plates.
- $y(Y-y)$ reflects the parabolic nature of the flow velocity, with zero velocity at the walls ($y = 0$ and $y = Y$) due to the no-slip condition and a maximum velocity at the channel center.

In the implementation, as shown in Listing 4 den_in and den_out represent the density at the inlet and outlet, creating a pressure difference across the channel. The function equilibrium_pois calculates the equilibrium distribution for Poiseuille flow using the local density d and velocity v. The function periodic_conditions apply periodic boundary conditions at the inlet and outlet to simulate an infinitely long channel. It uses the equilibrium distribution functions equi_in and equi_out for the inlet and outlet.

```python
import numpy as np
den_in = 1+diff
den_out = 1-diff
den=np.ones((x,y+2))
vel=np.zeros((2,x,y+2))
def equilibrium_pois(d,v):
    vel_mag =  v[0,:] ** 2 + v[1,:] ** 2
    temp = np.dot(v.T,c.T).T
    sq_velocity = temp ** 2
    f_eq =( ((1 + 3*(temp) + 9/2*(sq_velocity) - 3/2*(vel_mag)) * d).T * w
       ).T
    return f_eq

def periodic_conditions(den,vel,den_in,den_out,f):
        equi = equilibrium(den,vel)
        equi_in = equilibrium_pois(den_in, vel[:,-2,:])
        f[:, 0, :] = equi_in + (f[:, -2, :] - equi[:, -2, :])
        equi_out = equilibrium_pois(den_out, vel[:, 1, :])
        f[:, -1, :] = equi_out + (f[:, 1, :] - equi[:, 1, :])
        return f

f=equilibrium(den,vel)
for step in range(steps):
    den=density(f)
    vel=velcoity(f,den)
    f=periodic_conditions(den,vel,den_in,den_out,f)
    f=streaming(f)
    f=boundry_condition(f)
    f=collision(f,vel,den)
#rest of the code
# boundary conditions
def boundry_condition(f,lid_vel):
    f[2, :, 1] = f[4, :, 0]
    f[5, :, 1] = f[7, :, 0]
    f[6, :, 1] = f[8, :, 0]
    f[4, :, -2] = f[2, :, -1]
    f[7, :, -2] = f[5, :, -1] - 1 / 6 * lid_vel
    f[8, :, -2] = f[6, :, -1] + 1 / 6 * lid_vel
    return f
#rest of the code
delta = 2.0 * diff /x / shear_viscosity / 2.
yi = np.linspace(0, y, y+1) + 0.5
Analytical = delta * yi * (y - yi) / 3.
#rest of the code
```

Listing 4: Python code for the Poiseulle flow implementation

3.5 Sliding lid

A square cavity with a moving top lid simulates a complex flow pattern. In this configuration, the top boundary moves at a constant velocity U_w, while the remaining three walls remain stationary. The fluid velocity at the walls is zero. This setup induces circulation within the cavity. The motion of the sliding lid introduces turbulence into the cavity, a phenomenon characterized by the Reynolds number Re. The Reynolds number is a dimensionless quantity that is used to predict the flow pattern in different fluid flow situations. It is the ratio between inertial and viscous forces.

$$Re = \frac{lu}{\nu} \tag{9}$$

l is the side length of the cavity
u is the velocity of the of the lid
ν is the kinematic viscosity of the fluid inside the cavity

At low Reynolds numbers, the flow is steady and laminar and the vortex is in the center of the cavity. As the Reynolds number increases, the flow can become unsteady and transition to turbulent flow, with additional vortices appearing in the cavity. We also get the periodic bounce back from the left and right walls.

```python
import numpy as np

den=np.ones((length+2,length+2))
vel=np.zeros((2,length+2,length+2))
f=equilibrium(den,vel)
for step in range(steps):
    f=streaming(f)
    f=boundry_condition(f,lid_vel)
    den=density(f)
    vel=velcoity(f,den)
    f=collision(f,vel,den,relaxation)
#boundary condition for lid driven
def boundry_condition(f,lid_vel):
    # Top and bottom no-slip boundaries (moving lid at the top)
    f[2, :, 1] = f[4, :, 0]
    f[5, :, 1] = f[7, :, 0]
    f[6, :, 1] = f[8, :, 0]
    f[4, :, -2] = f[2, :, -1]
    f[7, :, -2] = f[5, :, -1] - 1 / 6 * lid_vel
    f[8, :, -2] = f[6, :, -1] + 1 / 6 * lid_vel
    f[[3,6,7],-2,:]=f[[1,8,5],-1,:]
    f[[1,5,8],1,:]=f[[3,7,6],0,:]
    return f

raynolads=1000
length=300
lid_vel=0.1
relaxation=((2 * raynolads) / (6 * length * lid_vel + raynolads))
steps=10000
rate=100
#rest of the code
```

Listing 5: Python code for the Sliding Lid implementation

c and w define the lattice velocities and weights, respectively, for the D2Q9 model. The function boundry_condition enforces boundary conditions for a lid-driven cavity. The no-slip boundary condition at the walls ensures the fluid sticks to the boundaries, which is implemented by mirroring opposite distribution functions and adjusting for the lid velocity. The cavity size (length), the lid velocity (lid_vel) and The Reynolds number (raynolads) determines the flow. This influences the relaxation parameter and the rate at which it returns to equilibrium.

3.6 Parallelization using MPI

Parallelization using the Message Passing Interface (MPI) is a method for running simulations on multiple processors. MPI provides a standard library for passing messages between nodes in a distributed-memory system, which is ideal for simulations that can be broken down into smaller, independent tasks [8].

The two main steps in LBM are the collision step, where particle distributions at each node are relaxed towards their equilibrium states, and the streaming step, where particles are propagated to adjacent nodes. Both steps can be executed parallelly with minimal inter-process communication. MPI is used to exchange the contents of ghost cells. Ghosts cells store the data at the boundary of a subdomain to perform the streaming step. Before the streaming step can proceed, the boundary data needs to be updated with information from neighboring subdomains. Since the collision step only involves local node data, it can be performed independently by each processor without the need for inter-processor communication. After the collision step, the streaming step requires each subdomain to have the updated boundary data from its neighbors. Here, MPI communication is essential to ensure that the occupation numbers are correct across the entire domain [8].

The ability to perform many operations in parallel, combined with efficient communication strategies, allows for good scalability. As more processors are added, the computation time should decrease proportionally. It is important to balance the computational load across processors and minimize communication overhead.

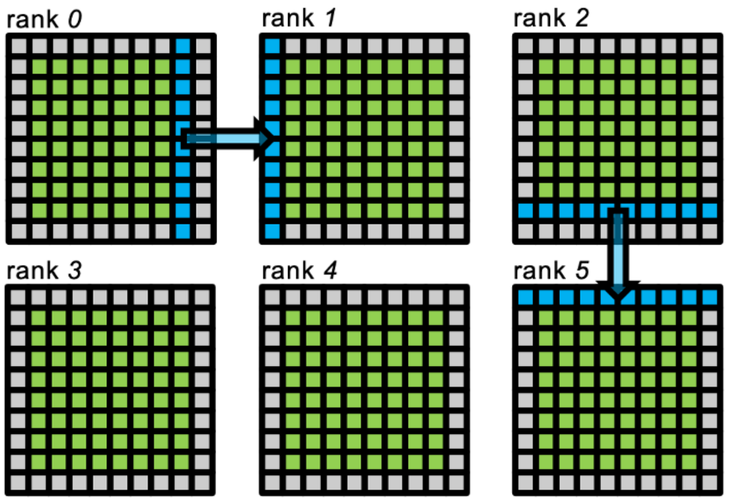


Figure 3: Domain decomposition and communication strategy [8]

A two-dimensional lattice is decomposed into a spatial domain of equal size. the grey lattice points are the ghost nodes. During each communication step, the outermost green lattice points are communicated with adjacent outermost grey lattice points [8]. Each process is assigned a unique identifier known as its "rank". Each square with a "rank" number represents a subdomain handled by a separate process in a parallel computation as illustrated in Figure 3

The communicate function partitions the computational domain across multiple processes. The function takes in the MPI object mpi, the field f representing the fluid's distribution function, boolean flags br, bl, bb, bt indicating the presence of right, left, bottom, and top boundaries respectively, and nr, nl, nb, nt which are the ranks of the neighboring processors. The boolean flags check whether the subdomain has a boundary at each direction. If it does not, it needs to communicate with the neighboring subdomain in that direction. Sendrecv function is called when there is a need to send and receive boundary data with the neighbor subdomains. The boundary data is first copied into a buffer, recvbuf, which is then filled with incoming data from the adjacent subdomain. After the communication, the ghost cells (recvbuf) in the field f are updated with the data received from the neighboring subdomains as shown in Listing 6.

```python
import numpy as np

#rest of the code
def communicate(mpi,f,br,bl,bb,bt,nr,nl,nb,nt):
        if not br:
            recvbuf = f[:, -1, :].copy()
            mpi.Sendrecv(f[:,-2, :].copy(), nr, recvbuf=recvbuf, sendtag =
                10, recvtag = 20)
            f[:, -1, :] = recvbuf
        if not bl:
            recvbuf = f[:, 0, :].copy()
            mpi.Sendrecv(f[:, 1, :].copy(), nl, recvbuf=recvbuf, sendtag =
                20, recvtag = 10)
            f[:, 0, :] = recvbuf
        if not bb:
            recvbuf = f[:,: ,0 ].copy()
            mpi.Sendrecv(f[:, :,1 ].copy(), nb, recvbuf=recvbuf, sendtag =
                30, recvtag = 40)
            f[:, :, 0] = recvbuf
        if not bt:
            recvbuf = f[:, :, -1].copy()
            mpi.Sendrecv(f[:, :, -2].copy(), nt, recvbuf=recvbuf, sendtag
                = 40, recvtag = 30)
            f[:, :, -1] = recvbuf

        return f

#rest of the code
```

Listing 6: Python code for Parallelization using MPI

3.7 Shear Wave Decay

Shear wave decay characterizes how perturbations in velocity or density diminish over time as a result of viscous effects. To obtain the analytical solution, sinusoidal perturbations are used as initial conditions and observed over time, as shown in Equation 10 and Equation 11.

The initial conditions at time $t = 0$ for a fluid dynamics simulation are given by the density and velocity distributions along the x-axis as follows [11]:

$$\rho(x, t = 0) = \rho_0 + \epsilon \sin\left(\frac{2\pi x}{L_x}\right) \tag{10}$$

where ρ_0 represents the baseline density, which is uniform across the domain in the absence of any perturbation.

The initial velocity distribution:

$$u_x(x, t = 0) = \epsilon \sin\left(\frac{2\pi x}{L_y}\right) \tag{11}$$

where ε is the small amplitude of perturbation, suggesting that the initial deviation from the baseline density is minimal.

Here, L_x and L_y denote the characteristic lengths of the domain along the x and y axes, respectively. The sine functions induce a periodic variation in the density and velocity fields, with wavelengths proportional to the domain sizes L_x and L_y.

The Navier Stokes equation for incompressible fluids:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} + \frac{1}{\rho}\nabla p = \nu \nabla^2 \mathbf{u} \tag{12}$$

We make assumptions to get an analytical solution:

1. Perturbation is small, velocity gradient is small therefore the non-linear term $(\mathbf{u} \cdot \nabla)\mathbf{u}$ can be neglected.

2. The pressure gradient ∇p is small and negligible.

This gives us the below analytical solution [**?**] :

$$a(t) = a_0 + \varepsilon \exp\left(- \left(\frac{2\pi\nu}{L} \right)^2 t \right) \tag{13}$$

where $a(t)$ represents the amplitude of the perturbation, a_0 is the initial amplitude, ε is the perturbation magnitude, ν is the kinematic viscosity of the fluid, L is a characteristic length scale, and t is time. .The exponential term with the negative sign in the exponent indicates that the perturbation amplitude decays over time.

4 Validation and Result

This section focuses on validating and evaluating the scalability of the implemented Lattice Boltzmann Method (LBM) for simulating fluid dynamics. Four scenarios: shear wave decay, Couette flow, Poiseuille flow, and lid-driven cavity are investigated to assess the accuracy and performance of the implementation. The results are presented, including comparisons with analytical solutions and an analysis of the LBM's parallelization performance.

4.1 Couette Flow

In Couette flow, the bottom wall is rigid which is fixed and the moving wall is at the top and moves to the right as shown in Figure 2.

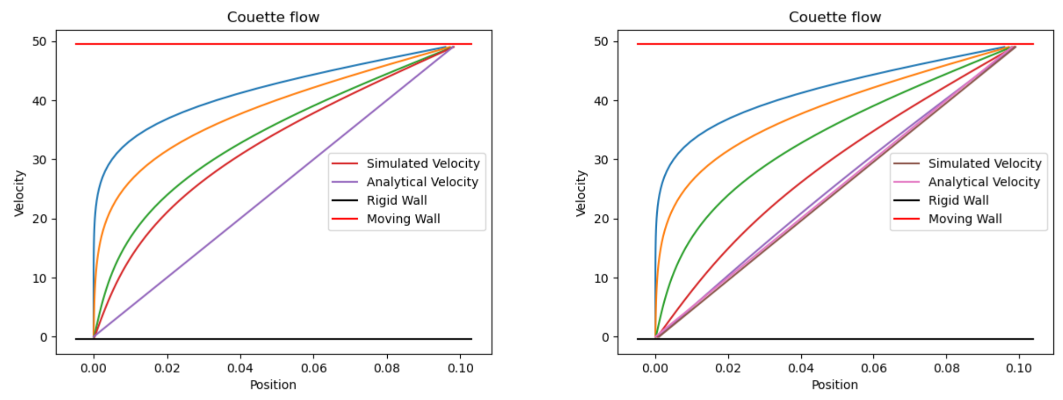


Figure 4: Couette flow: Left- for step size 500 Right- for step size 3000, converging towards analytical solution

Figure 4 shows the simulation result, the top red wall is the moving wall and the bottom black wall is the rigid wall. Initially, the velocity profile has the highest values at the moving wall when the step size is 500. Over time, the flow propagates to the bottom meaning when the step size is increased to 3000 it converges to the analytical solution according to Equation 8.

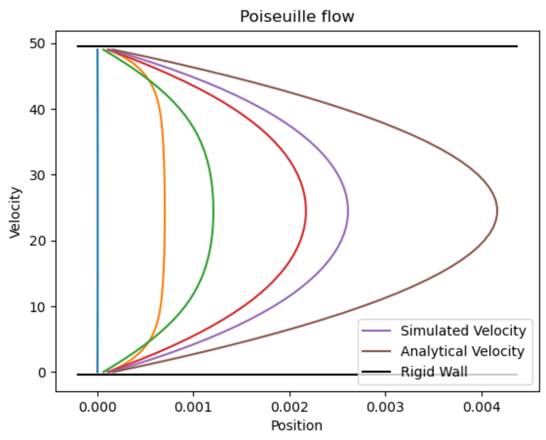
The results are obtained for lattice size 100x50, lid velocity $U_w = 0.1$, relaxation rate $\omega = 0.5$, and simulated for 3000 steps.

4.2 Poiseuille Flow

In Poiseuille Flow both top and bottom boundaries have rigid walls and the inlet and outlet are implemented using PBCs with pressure gradient illustrated in Figure 2.

The analytical solution is given in Equation 8. Figure 5 compares the analytical solution and simulated result. The result shows how the velocity profile converges to an analytical solution with time. This can be observed in Figure 5 with step size 1000 when the simulated result deviates largely from the analytical solution, but with time the flow converges to an analytical solution with step size 5000.

The results are obtained for a 100x50 lattice grid, relaxation rate $\omega = 0.5$, and simulated for 3000 steps.



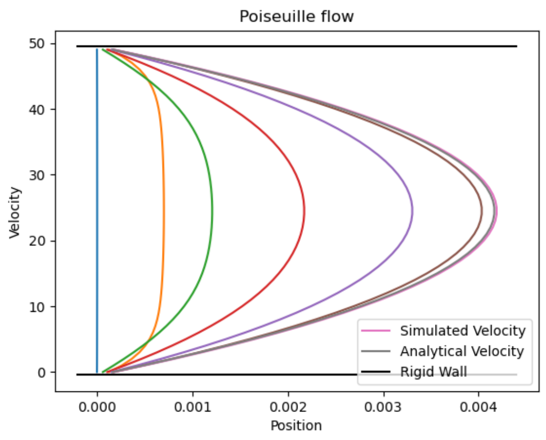


Figure 5: Poiseuille flow converging towards the analytical solution. The image on the left shows the simulation result with step size=1000, and Right side with step size=3000

4.3 Sliding Lid

In a sliding lid, the top boundary moves at a constant velocity U_w while the other three walls remain stationary, creating circulation within the cavity. The sliding lid induces turbulence in the lid cavity. The relaxation constant is calculated with Raynolds number and lid velocity.

Figure 6 shows the step by step flow for various time steps, for the lattice grid 300x300, Reynolds number =1000 for steps 100,200,400,800,1600,3200,6400,10000 respectively. The final image is the Sliding lid after 10000 steps where we can see the turbulence in the lid cavity.

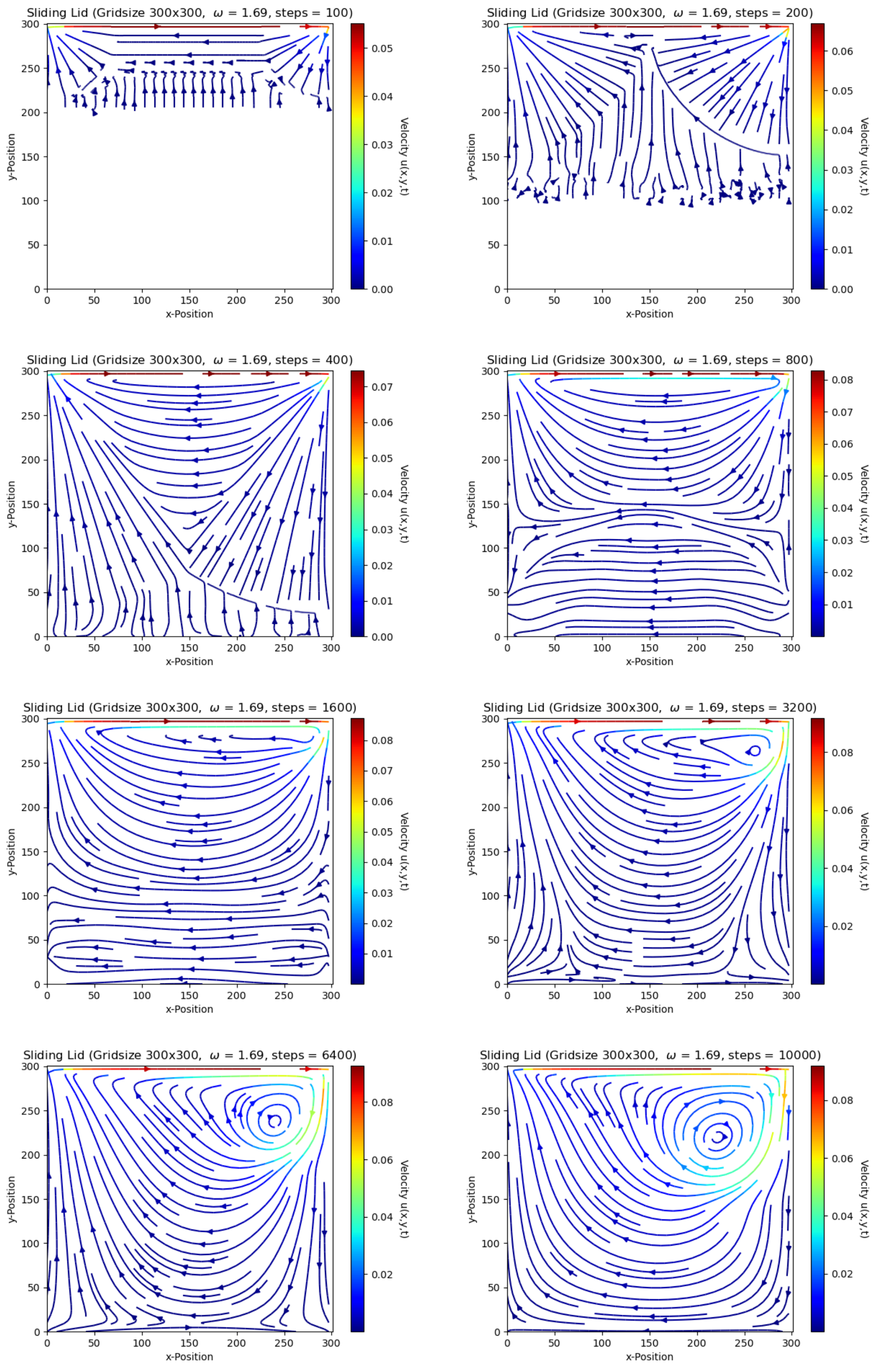


Figure 6: Sliding lid with grid size 300x300 for steps up to 10000 with relaxation ω=1.69

4.4 Parallelization

The sliding lid implementation was run on different lattice sizes and different processors in Parallelization using MPI and two-dimensional decomposition. This is done to check the scalability of the parallel implementation and the tests were run on bwUniCluster. The results are compared using MLUPS, the number of million lattice updates per second.

The simulation is run on 400 processors for 10000 steps. Figure 7 shows a logarithmic graph comparing the performance of a Lattice Boltzmann simulation in terms of MLUPS (Million Lattice Updates Per Second) against the number of processors used for different grid sizes. The X-axis, which is on a logarithmic scale, shows the number of processors, while the Y-axis, also logarithmic, shows the MLUPS. Each line represents a different grid size, varying from 200x200 to 1000x1000.

It is observed that as the number of processors increases, the MLUPS increases as well, indicating better performance. This is expected as more processors should be able to handle more lattice updates in parallel, thus completing more updates per second. The different lines show that larger grid sizes generally achieve higher MLUPS, which suggests that the simulation can utilize the additional processors more efficiently with larger amounts of data. It is also observed that beyond a certain number of processors, the efficiency gain decreases, and this is where the curves start to flatten. This flattening indicates that adding more processors does not significantly increase performance, this is likely due to the communication overhead between processors.

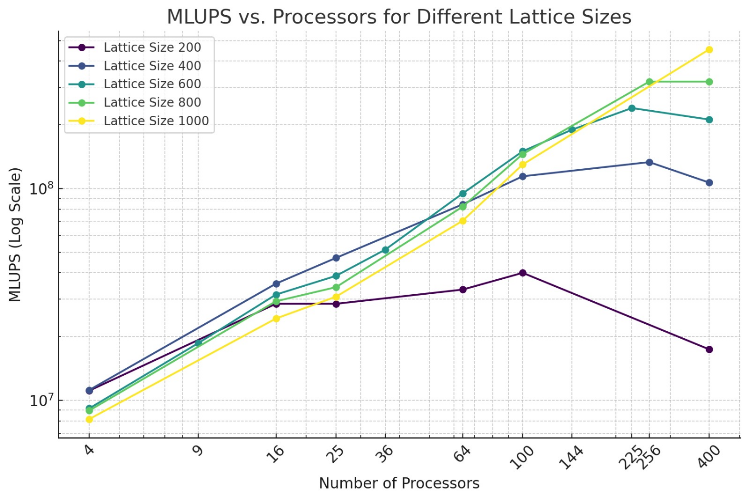


Figure 7: Lattice Boltzmann implementation as a function of the number of CPUs (MPI processes)

5 Conclusion

In this report, we discuss the Lattice Boltzmann method and its implementation in computational fluid dynamics. We established a solid understanding of LBM fundamentals, including discretization techniques and boundary-handling methods. Validation experiments, like shear wave decay and Couette flow, confirmed the accuracy of our simulations against theoretical predictions.

We discussed the algorithmic implementation, emphasizing the use of numpy for Python-based simulations and MPI for load balancing through domain decomposition. Validation involved comparing simulation results with analytical solutions. The lid-driven cavity simulation demonstrated LBM's ability to model fluid dynamics across different Reynolds numbers and turbulent flow inside the cavity.

Scalability tests on the bwUniCluster showed promising linear scalability up to a certain threshold, beyond which communication overhead limited further gains. This emphasizes the importance of optimizing communication strategies for efficient parallel computing.

This indicates the correctness of the LBM implementation and showcases its potential for complex fluid dynamics simulations in high-performance computing. Future work could explore optimization techniques, more complex boundary conditions, and extending LBM to three-dimensional simulations, widening its applicability in fluid mechanics and beyond.

References

[1] *Anderson, J. D. (1995)"Computational Fluid Dynamics: The Basics with Applications". McGraw-Hill*

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

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

[4] *Qian, Y. H., d'Humières, D., Lallemand, P. (1992). Lattice BGK Models for Navier-Stokes Equation. Europhysics Letters, 17(6), 479.*

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

[6] *"Lattice Boltzmann methods," Wikipedia [Online]. Available: https://en.wikipedia.org/wiki/Lattice_Boltzmann_methods . [Accessed: 20-03-2024].*

[7] *Kutay, M. Emin, "Lattice Boltzmann Resources" [Online]. Available: https://www.egr.msu.edu/~kutay/LBsite/index.html . [Accessed: 20-03-2024].*

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

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

[10] *Fully Developed Couette and Poiseulle Flows [Online]. Available: https://thestemtutor.org/?p=7099/ [Accessed: 20-03-2024]*

[11] *Batchelor GK. Frontmatter. In: An Introduction to Fluid Dynamics. Cambridge Mathematical Library. Cambridge University Press; 2000:i-iv.*