High Performance Computing with Python

Final Report

ROUNACK KUNDU

5583891

kundur@informatik.uni-freiburg.de

Professor and Instructor - **Lars Pastewka** & **Andreas Greiner**

February 15, 2024

Contents

1

Introduction

High-Performance Computing (HPC) has revolutionized a number of scientific and engineering domains by making it possible to simulate complicated processes with state of the art accuracy, efficiency and speed. One such method in the field of computational fluid dynamics is that of the Boltzmann Lattice Method (LBM). This report is structured around the "High High-Performance Computing: Fluid Mechanics with Python" course offered by the University of Freiburg ". The main aim of the course is to implement and parallelize the Lattice Boltzmann Method.

This method is particularly robust for modeling in fluid flow applications involving interfacial dynamics and complex boundaries and fluid-structure interactions. Unlike conventional numerical schemes based on discretizations of macroscopic continuum equations, the lattice Boltzmann method is based on microscopic models and mesoscopic kinetic equations.[1] Also, unlike conventional Navier-Stokes solvers, LBM is based on discretization principle and works on discreet velocities making it suitable for parallelization. The fundamental idea of the LBM is to construct simplified kinetic models that incorporate the essential physics of microscopic or mesoscopic processes so that the macroscopic averaged properties obey the desired macroscopic equations. [2]
In Simpler terms, Boltzmann's kinetic theory of gases formed the base for the Boltzmann transport equation, which is discretized by the LBM. "The LBM solves a discrete Boltzmann equation which is designed to reproduce the Navier-Stokes (N-S) equations in the macroscopic limit". [3]

1.1 Setup

For the whole simulation process, we divide the grid into lattices in the LBM process and the lattice nodes represent single particles. The number of possible connections and directions of these particle movements depends on the lattice arrangements.[4] The lattice arrangement which we use here is the D2Q9 which represents a discretized two-dimensional space with nine channels (one for each direction) as shown in figure 1.1.

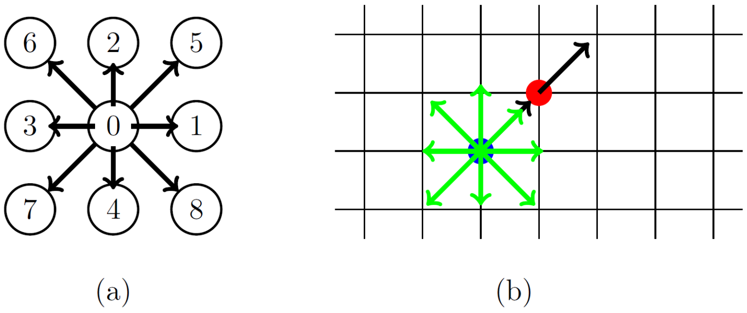

Figure 1.1: Discretization of the BTE. (a) Discretization on the velocity space according to D2Q9.(b) Uniform 2D grid for the discretization in the physical space1.

1.2 Technical Specifications

Most of the experiments are conducted on a local device except the parallelization steps for the Sliding Lid Problem. The local system has an Intel core i5-10300H (4 cores, 8 threads) CPU max clocking at @4.50 GHz with 8 GB of DDR4 RAM with an Nvidia GeForce GTX 1650 Ti 4 GB of GPU.

2

Methods

2.1 Lattice Boltzmann Method: Different Equations

The Lattice Boltzmann method is a very powerful approach to simulate fluid flows. In this section, we explore the fundamental equations underlying LBM and the different techniques used.

2.1.1 Boltzmann Transport Equation and Streaming

The Boltzmann transport equation can be formulated as:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_r f + \mathbf{a} \cdot \nabla_v f = C(f) \tag{2.1}$$

To simplify the understanding procedure, this equation can be written in terms of the dependant variables of each term also which was provided during the lecture as:

$$\frac{\partial f(\mathbf{r}, \mathbf{v}, t)}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{r}} f(\mathbf{r}, \mathbf{v}, t) + a f(\mathbf{r}, \mathbf{v}, t) = C(f(\mathbf{r}, \mathbf{v}, t)) \tag{2.2}$$

The probability density function f or the distribution function is the unknown and is calculated. It is a function of physical space r, time t and the velocity v. The l.h.s of the equation 2.2 is known as the streaming part and the r.h.s is known as the collision part. As we can see from the equation that the function f is dependant on the velocity v also along with time t and space r and so it needs to be discretized. It is discretized using finite differences keeping in mind the relaxations shown in fig 1.1. The discretized streaming equation is given by :

$$f_i(r + ci\Delta t, t + \Delta t) = f_i(r, t) + C(r, t) \tag{2.3}$$

Streaming can be described as the movement of particles in vacuum without considering the presence of other particles for it to interact with.[5] The velocity space can be discretized by directing the particle to move along predefined directions denoted as c_i, referred to as "velocity sets". In a 2D scenario illustrated in Figure 1.1 a, there are 9 possible directions considered for discretization. Consequently, we can represent the function $f(r, v, t)$ as $f_i(r, t)$, where the index i corresponds to the directions depicted in Figure 1.1 a. The time steps are discretized as Δt during which the f_i values are propagated. The velocity set is given by :

$$c = \begin{pmatrix} 0 & 0 & -1 & 0 & 1 & -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \end{pmatrix} \tag{2.4}$$

2.1.2 Collision Step

The r.h.s of equation 2.2 depicts the collision step. The collision step represents the particles interaction with each other within the lattice.[5] This is generally a two-part scattering integral and is a complicated task. Hence, we simplify it by approximating this term with the help of a relaxation constant τ :

$$\frac{\partial f(r, v, t)}{\partial t} = -\frac{f(r, v, t) - f_{\text{eq}}(r, v, t)}{\tau} \tag{2.5}$$

The discretized form of the collision part for time time step δt is given by :

$$C[f(r, v, t)] = -\omega[f_i(r, t) - f_{\text{eq}_i}(r, t)] \tag{2.6}$$

where C represents the collision term and ω is a constant.

The next part is to find out what the equilibrium distribution is in equation 2.6. The equilibrium distribution depends on the local density $\rho(r)$ and the local average velocity $u(r)$:

$$f_{\text{eq}_i}(\rho(r), u(r)) = w_i \rho(r) \left[1 + 3c_i \cdot u(r) + \frac{9}{2}(c_i \cdot u(r))^2 - \frac{3}{2}|u(r)|^2\right] \tag{2.7}$$

The dependant local quantities are calculated as :

$$\rho(r) = \sum_i f_i(3)$$
$$u(r) = \frac{1}{\rho(r)} \sum_i c_i f_i(r)$$

The w_i for a D2Q9 lattice can be defined as :

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

2.2 Boundary Handling

The boundary handling is an important aspect of the LBM. It helps in understanding the interaction of the particles inside the lattice with the boundary of the lattice under different boundary types and conditions.In Lattice Boltzmann Method (LBM), there are two primary arrangements for dealing with boundaries:

- **Dry node**: Boundaries are situated at the link between the nodes. To simplify, the boundaries are placed on the links connecting adjacent lattice nodes.
- **Wet node**: Boundaries are situated at the lattice points themselves.In other words, the boundaries coincide with the lattice points of the fluid grid.

For this implementation, our focus is on the **dry node** arrangement. The different boundary conditions that we encounter here are : [4][6]

2.2.1 Rigid Wall

The rigid wall boundary condition, as its name implies, employs a condition that is used to generally simulate bounce-back scheme within the system, effectively disrupting its periodicity in practice while theoretically retaining periodic behavior. The no-slip condition at the boundary is enforced, which dictates that fluid particles adhere to the boundary's velocity. When a particle collides with the boundary, its velocity is reversed in a bounce-back manner, effectively simulating reflection from the wall. This process can be expressed generally as:

$$\overline{f_i}(x_b, t + \Delta t) = f_i^*(x_b, t) \tag{2.9}$$

Here, $\overline{f_i}$ denotes the population of the opposite channels, and f_i^* represents the pre-streaming population.

2.2.2 Moving Wall

The moving wall boundary condition is very similar to the rigid wall condition. In moving wall, if the boundary is moving with a velocity U_w, the equation in 2.9 is modified to consider the variation in the moments of particles:

$$f_i(x_b, t + \Delta t) = f_i^*(x_b, t) - \frac{2 w_i \rho_w c_i \cdot U_w}{c_s^2} \tag{2.10}$$

2.2.3 Periodic Boundary

The periodic boundary condition is one which ensures continuous flow of particles i.e. particles flowing out of the domain on one side shall re-enter the domain on the opposite side.[7] This is used in cases where we want to check a finite section of the flow. It is given by :

$$f_i(x_1, t) = f_i(x_n, t) \tag{2.11}$$

2.2.4 Periodic boundary with pressure gradient

This is a modification of the above mentioned periodic boundary where there is a pressure variation between the inlet and the outlet. The variation in pressure is also related to the fluid's density between the boundaries. The equilibrium state is given by :

$$f_{\text{eq}}(x_0, y, t) = f_{\text{eq}_i}(\rho_{\text{in}}, u_N) \tag{2.12}$$

$$f_{\text{eq}}(x_{N+1}, y, t) = f_{\text{eq}_i}(\rho_{\text{out}}, u_1) \tag{2.13}$$

The inlet and outlet boundary conditions is finally given by :

$$f_i^*(x_0, y, t) = f_{\text{eq}_i}(\rho_{\text{in}}, u_N) + (f_i^*(x_N, y, t) - f_{\text{eq}_i}(x_N, y, t)) \tag{2.14}$$

$$f_i^*(x_{N+1}, y, t) = f_{\text{eq}_i}(\rho_{\text{out}}, u_1) + (f_i^*(x_1, y, t) - f_{\text{eq}_i}(x_1, y, t)) \tag{2.15}$$

The following are few illustrations of the boundary conditions:

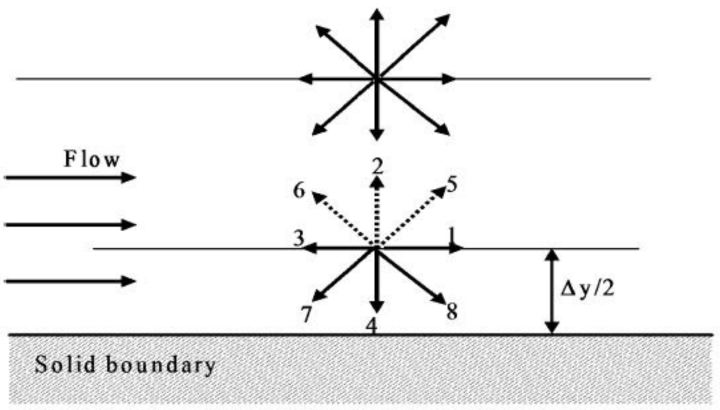

Figure 2.1: Scheme describing the Bounce Back Method (Mohamad, 2011).

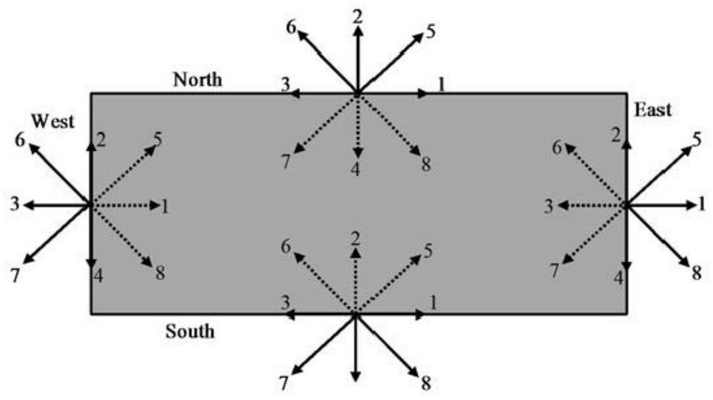

Figure 2.2: Schematic showing the known and unknown distribution functions at respective boundaries of the domain (Mohamad, 2011).

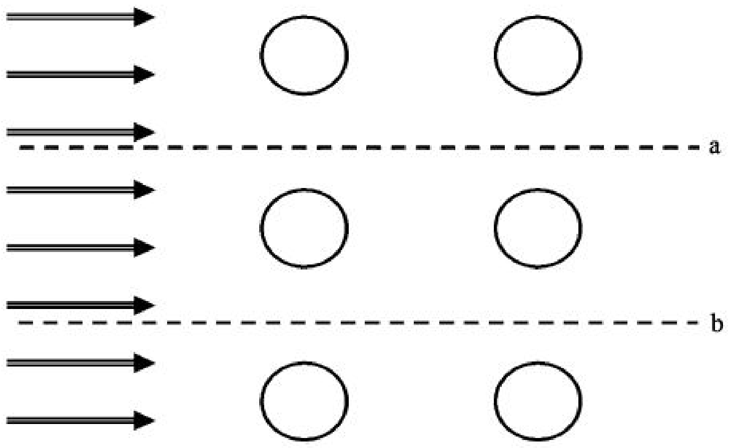

Figure 2.3: Schematic illustrating the periodic boundary conditions and how they relate to each other and the flow (Mohamad, 2011).

3

Implementation

The implementation is a modularized approach and has all the steps of the LBM in a systematic way. In this implementation, the main focus lies on how the fundamental equations of LBM are translated into Python code to simulate fluid dynamics correctly. The lattice grid is defined with the dimensions from the array lattice-grid-shape. The density matrix matches the grid shape while the velocity matrix is defined by ((2, lattice-grid-shape[0], lattice-grid-shape[1])). F can be calculated as a matrix of shape ((9, lattice-grid-shape[0], lattice-grid-shape[1])).

3.1 Streaming

The streaming step is handled by the streaming function which is described as :

Listing 3.1: Streaming step of the LBM

```python
def streaming(self) -> None:
    """
    Streaming step of the LBM
    """
    for i in range(9):
        self.f[i] = np.roll(self.f[i], shift=self.c_ai.T[i],
                            axis=[0, 1])
```

The streaming is handled by the inbuilt np.roll function. This function is used to roll array elements along a given axis.

3.2 Collision

The collision part is handled by first passing the pdf and the velocity matrix to calculate the density and the updated velocity at each time step . The collision function calls the equilibrium function within itself and then updates the states after the collision operation takes place. It is given in code by :

Listing 3.2: Density Update

```
def update_rho(self) -> None:
    """
        Update the density field
    """

    self.rho = np.einsum('ijk->jk', self.f)
```

Listing 3.3: Velocity Update

```
def update_u(self) -> None:
    """
        Update the velocity field
    """

    self.u = np.einsum('ij,jkl->ikl', self.c_ai, self.f) / self.rho
```

Listing 3.4: Collision Step

```
def collision(self) -> np.ndarray:
    """
        Collision step of the LBM
    """
    feq_inm = self.equilibrium_dist()
    return self.omega * (feq_inm - self.f)
```

Listing 3.5: Equilibrium

```
def equilibrium_dist(self) -> np.ndarray:
    """
        Calculate the equilibrium distribution function
    """
    cu = np.einsum('ai,anm->inm', self.c_ai, self.u)
    sq_cu = cu ** 2
    u2 = np.einsum('ijk,ijk->jk', self.u, self.u)
    w_rho = np.einsum('i,jk->ijk', self.w_i, self.rho)
    feq_inm = w_rho * (1 + 3 * cu + 4.5 * sq_cu - 1.5 * u2)
    return feq_inm
```

The np.einsum function is used here to perform the summation in the matrices along a particular axis. This is an inbuilt function in Numpy package.

3.3 Boundary Handling

Here, we discuss the implementation of the various boundary conditions and it's handling. The periodicity of the boundary can be implemented implicitly by the np.roll function, but we still need to store the pre-streaming state to apply the condition of boundary handling and then suitable change the state to simulate the new current condition and for this we use a dummy variable. To determine which boundaries use the rigid or moving walls, we define arrays which store the channel numbers based on Fig 2.1 and Fig 2.2 (example of boundary numbers), and depending on which boundary it is, we use the corresponding arrays. There are four arrays for four directions and each array stores three integers.

Listing 3.6: Boundary arrays

```
'''The rigid_index[] contains the directions bottom,
bot-left and bot-right and the mirrored
directions in opp[] array'''

    rigid_index = np.array([4, 7, 8])
    rigid_opp = np.array([2, 5, 6])

'''The moving_index[] contains the directions top,
top-left and top-right and the mirrored
directions in opp[] array'''

    moving_index = np.array([2, 5, 6])
    moving_opp = np.array([4, 7, 8])
```

The following listing shows an example of a rigid boundary wall implementation :

Listing 3.7: Rigid Wall Boundary

```
'''checking if we are at the south boundary and
applying rigid wall'''
    if boundary[2] == True:
        if i in index_south:
            f_inm[index_north[np.where(i == index_south)
            [0][0]], -1, :] = f_inm_dummy[i, -1, :]
```

Here the array marked by **index_south** represents points/ channels facing bottom. This is an implementation of a rigid wall condition in the bottom boundary. The **f_inm_dummy** is the dummy variable to store the pre-streaming step.

Listing 3.8: Moving Wall Boundary

```
'''checking if we are at the south boundary and
applying rigid wall'''
    if boundary[3] == True:
        if i in index_north:
            '''applying moving wall at the
            north boundary'''
            f_inm[index_south[np.where(i == index_north)
            [0][0]], 0, :] = f_inm_dummy[i, 0, :] - 2 *
            w_i[i] * rho_average * ((c_ai[:, i] @ u_w).T
            / (c_s ** 2))
```

This is an implementation of the moving wall where **u_w** is the wall velocity that is specified by us.
The periodic boundary with pressure variation has a slight different implementation. We do not input the pressure variation directly and so we have to calculate the difference from the inlet and outlet densities.

Listing 3.9: Pressure Variation

```
'''calculate inlet pdf'''
    f_eq_in = self.equilibrium_dist_pdfparam(rho_inlet,
    u_out)
    f_int_beg = self.f[:, :, -2]
     - self.equilibrium_dist_pdfparam(self.get_rho()[:, -2],
     u_out)
    self.f[:, :, 0] = f_eq_in + f_int_beg

 '''calculating outlet pdf'''
    f_eq_out = self.equilibrium_dist_pdfparam(rho_outlet,
    u_in)
    f_int_end = self.f[:, :, 1]
     - self.equilibrium_dist_pdfparam(self.get_rho()[:, 1],
     u_in)
    self.f[:, :, -1] = f_eq_out + f_int_end
```

The **rho_inlet** is the density at the inlet and similarly **rho_outlet** is the density at the outlet. We add intermediate terms at the beginning and end to enhance the code understanding and also the equilibrium calculation is a modified version of Listing 3.5 .

3.4 Parallelization

To efficiently handle large-scale simulations and processes, LBM can be parallelized using spatial domain decomposition and the Message Passing Interface (MPI). This approach divides the simulation domain into smaller subdomains, distributing the computational load across multiple MPI processes. We decompose spatial domain using the message passing interface MPI.[5] To achieve this, we use the cartesian communicator from MPI, initialized as cartcomm in the code. The domain is partitioned into smaller subdomains according to the specified number of processes in the x and y directions. Each subdomain is assigned a unique rank, and autonomous computation occurs within these subdomains. Although parallelism is inherent in the collision step, communication plays a vital role during the streaming step, where particles migrate between neighboring subdomains. To handle this, additional layers of cells known as "ghost regions" are introduced around each subdomain. These ghost cells retain pertinent data from adjacent processes, facilitating precise data exchange across domain borders. The cartesian communicator implementation is based on the example provided during the lecture:

Listing 3.10: Cartesian Communicator

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

    '''Send to left and receive from right'''
    sendbuf = np.ascontiguousarray(c[:, :, 1])
    recvbuf = c[:, :, -1].copy()
    cartcomm.Sendrecv(sendbuf=sendbuf,
        dest=dL, recvbuf=recvbuf, source=sL)
    c[:, :, -1] = recvbuf
    ''''''

    '''Send to right and receive from left'''
    sendbuf = np.ascontiguousarray(c[:, :, -2])
    # Send the second last column to dR
    recvbuf = c[:, :, 0].copy()
    cartcomm.Sendrecv(sendbuf=sendbuf,
        dest=dR, recvbuf=recvbuf, source=sR)
    c[:, :, 0] = recvbuf
    # received into the 0th column from sR
    ''''''

    '''Send to down and receive from up'''
    sendbuf = np.ascontiguousarray(c[:, -2, :])
    recvbuf = c[:, 0, :].copy()
    cartcomm.Sendrecv(sendbuf=sendbuf,
        dest=dD, recvbuf=recvbuf, source=sD)
```

```
c[:, 0, :] = recvbuf
''''''

'''Send to up and receive from down'''
sendbuf = np.ascontiguousarray(c[:, 1, :])
recvbuf = c[:, -1, :].copy()
cartcomm.Sendrecv(sendbuf=sendbuf,
        dest=dU, recvbuf=recvbuf, source=sU)
c[:, -1, :] = recvbuf
''''''

return c
```

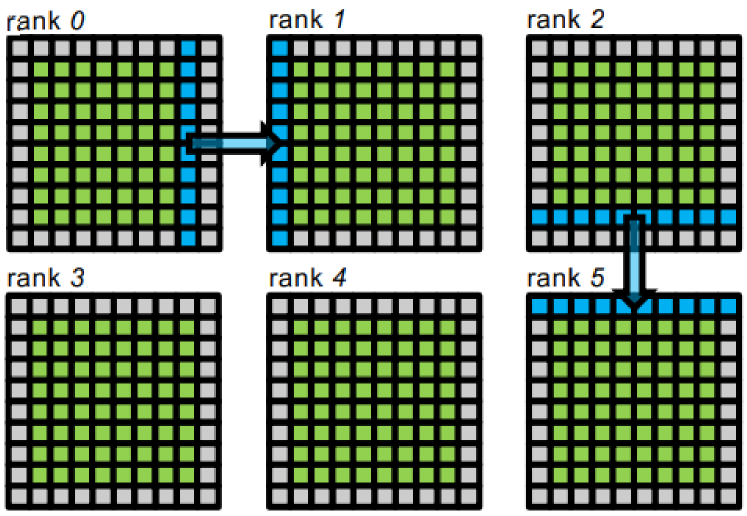

Figure 3.1: Decomposition and Communication Strategy. The whole two-dimensional lattice is decomposed into spatial domains of roughly equal size(green lattice points). These green lattice points form the active physical domain for each rank. We introduce extra ghost cells, represented by grey lattice points, to serve as buffers. During communication, outermost green lattice points send data to adjacent outermost ghost points (blue arrows). The process requires four communication steps, out of which two of which are indicated by the arrows.[5]

4

Experiments and Results

In this part, we validate the implementation of our LBM by conducting a series of experiments and also provide some numerical and visual findings.

4.1 Shear Wave Decay

The shear wave decay experiment illustrates how a disturbance in a sinusoidal shear wave influences the evolution of density and velocity until reaching a steady state. The convergence to this stable condition is facilitated by viscosity, which is determined by the parameter ω . As viscosity diminishes flow velocity, it eventually converges to zero.
To derive an analytical description of this phenomenon, we start with the Navier-Stokes equations given by:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u + \frac{1}{\rho}\nabla p = \nu \nabla^2 u \tag{4.1}$$

However, due to the absence of an analytical solution, we make two assumptions to get close to an analytical solution:

1. Given the small perturbation, we consider the velocity gradient negligible enough to disregard the non-linear term $(u \cdot \nabla)u$.
2. The pressure gradient Δp is also negligible due to its small magnitude.

With these assumptions, we obtain the analytical solution used for validation in our experiments:[8]

$$a_t = a_0 + \epsilon \exp\left(-\nu \left(\frac{2\pi x}{L}\right)^2\right) \tag{4.2}$$

This equation characterizes the exponential decay of density or velocity following an initial perturbation, as elaborated in the subsequent subsections. Here, a_0 represents the amplitude, set to the initial density for density

decay or zero for velocity decay. The viscosity, denoted by ν, is dependent on ω through the analytical formula $c_s^2(1/\omega - 1/2)$.
The following density and velocity wave conditions were used during the experiment. The initial density at $t = 0$ follows the formula:

$$\rho(r, 0) = \rho_0 + \epsilon \sin \frac{2\pi x}{L_x} \quad (4.3)$$

Here, $0 < \rho_0 < 1$ represents the initial density, and L_x denotes the domain length in the x direction. The equation introduces a sinusoidal wave perturbation, allowing us to observe the density profile's evolution over time. ϵ signifies the wave's amplitude.

For the velocity shear wave decay scenario in Fig. 4.4, the initial velocity u_0 at $t = 0$ is initialized as:

$$u(r, 0) = \epsilon \sin \frac{2\pi x}{L_y}$$

The experiment is run on a 100X100 lattice for 10,000 timesteps with ϵ 0.01 , initial ρ 0.5 and ω 1.0. These were the observations :

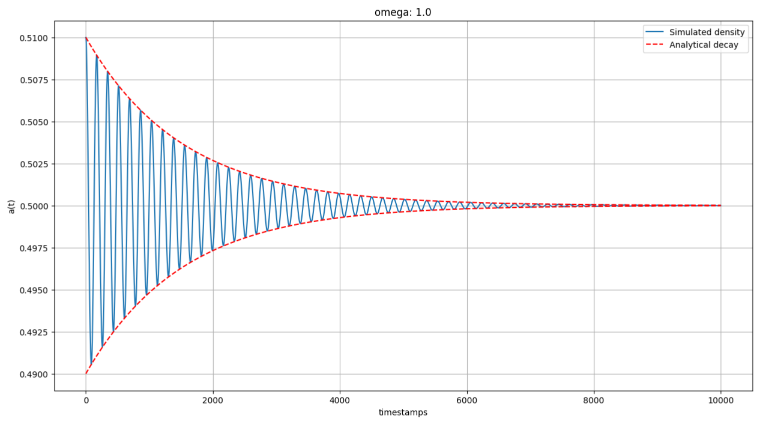

Figure 4.1: This shows the comparison between analytical density decay and the simulated decay .

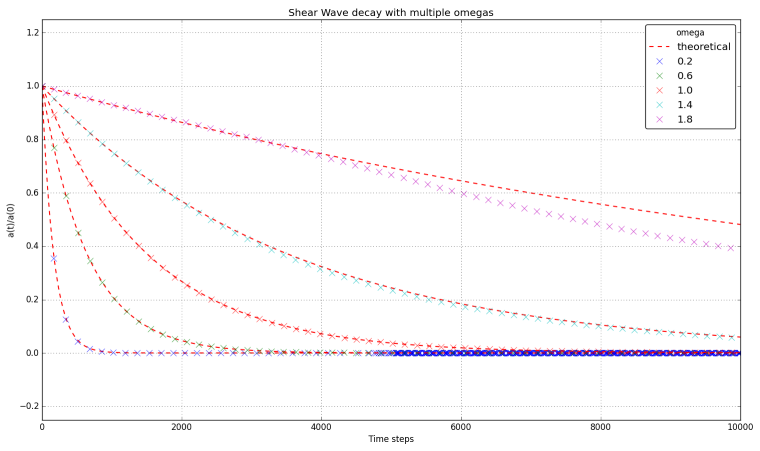

Figure 4.2: Time evolution of the amplitude of perturbation for different ω vs their theoretical values.

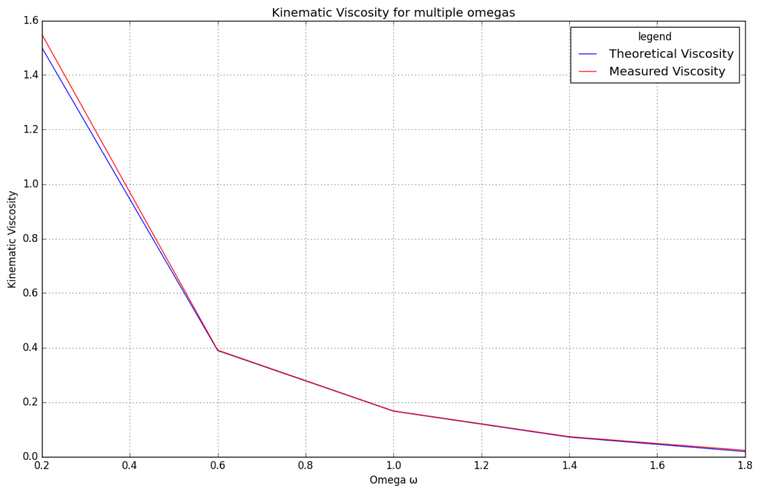

Figure 4.3: Theoretical Kinematic Viscosity vs Simulated Kinematic Viscosity for different ω.

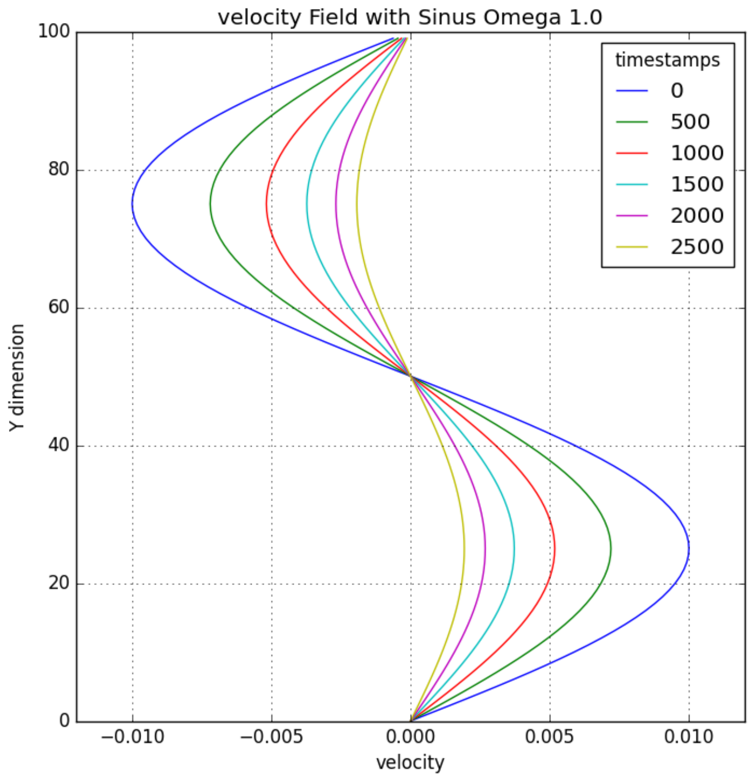

Figure 4.4: The time evolution of the simulated sinusoidal velocity with lattice grid size of (100, 100). The plot also shows the analytical decay up-to timestamps of 2500 which are divided for better understand-ability.

4.2 Couette Flow

In fluid dynamics, Couette flow is the flow of a viscous fluid in the space between two surfaces, one of which is moving tangentially relative to the other. Here, we will implement a moving wall. In this scenario the fluid flows between a fixed and a moving wall. The motion occurs because of the application of a viscous drag force on the fluid. In this simulation, we employ bounce-back boundary conditions at both the moving and stationary walls, along with periodic boundary conditions at the inlet and outlet. The analytical solution for this setup allows us to validate the dynamic wall implementation. The analytical expression is given by [9]:

$$u_x(\cdot, y) = \frac{Y - y}{Y} U_w$$

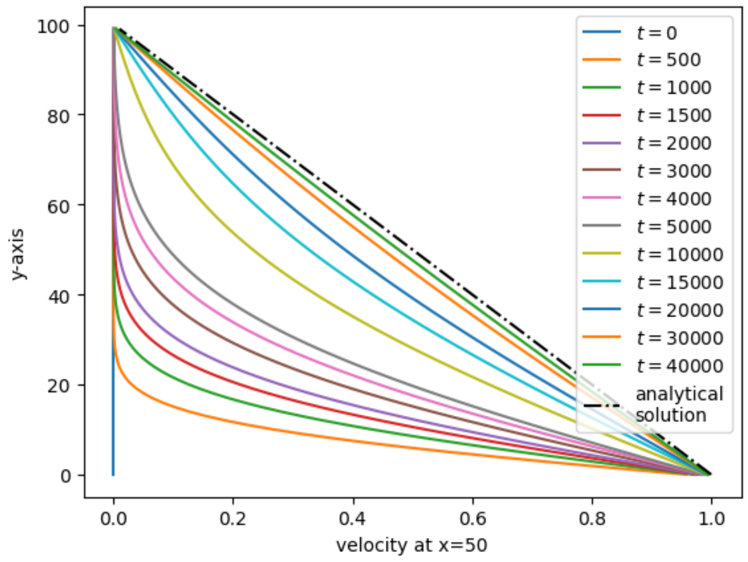

Figure 4.5: Velocity profile for Couette flow at staggered time-step intervals. Lattice grid size is 100x100 measured at x=50, ω=1.3 and measured over 50000 time-steps

4.3 Poiseuille Flow

Poiseuille flow, a variant of Couette flow, employs periodic boundaries with pressure fluctuations at the left and right boundaries. It is the pressure gradient that triggers a flow resembling that of a pipe. An analytical solution exists for the velocity field of this pipe flow, known as the Hagen-Poiseuille flow. Under the assumption of laminar flow, we can simplify the Navier-Stokes equations and consider only the streamwise component of the x axis, which is aligned with the pipe axis[10]:

$$\frac{\partial p(x)}{\partial x} = \frac{1}{\mu}\frac{\partial^2 u_x(y)}{\partial y^2}.$$

Thanks to the above assumption, we can integrate the above equation along the wall-normal direction y twice and, using the boundary conditions $u(0) = 0$ and $u(h) = 0$, we obtain the velocity profile:

$$u(y) = -\frac{1}{2\mu}\frac{dp}{dx}(h - y).$$

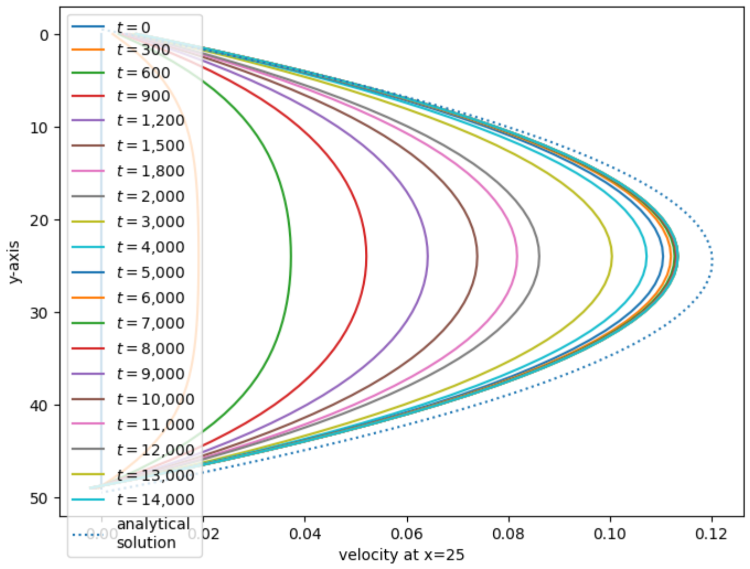

Figure 4.6: The velocity vectors for Poiseuille Flow at $x = 25$, for a lattice grid size of $(50, 50)$ at time step $= 15000$. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. The relaxation term $\omega = 1.0$. The inlet and outlet density are $\rho_{\text{in}} = 1.005$ and $\rho_{\text{out}} = 0.995$ respectively.

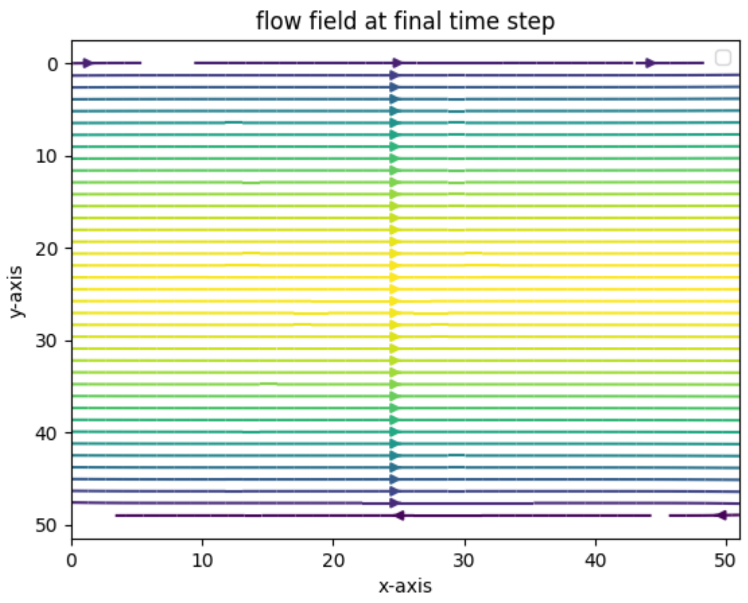

Figure 4.7: The velocity Flow Field at final time step

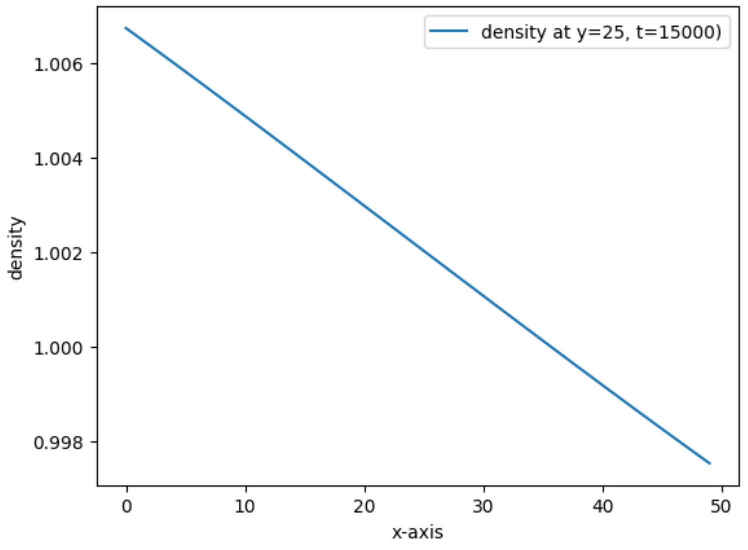

Figure 4.8: The Density Gradient at center Y=25

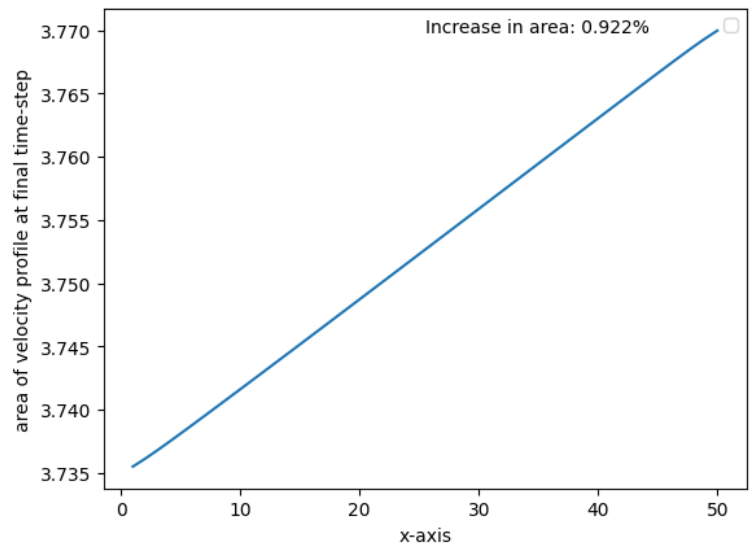

Figure 4.9: Area of velocity profile from left to right at final time step

4.4 The Sliding Lid : Serial and Parallel implementation

The lid-driven cavity test is frequently used in fluid dynamic simulations [5]. This scenario involves a two-dimensional fluid flow within a closed container, propelled by a lid in motion positioned at the top.[11] The other walls of the container and rigid and employ a bounce-back situation wheres the top wall or in this case, the lid moves with a certain velocity. One important concept introduced here is that of Reynolds Number. The Reynolds number is the ratio of inertial forces to viscous forces within a fluid that is subjected to relative internal movement due to different fluid velocities.[12] The Reynolds number is defined as:

$$\mathrm{Re} = \frac{uL}{\nu}$$

where:

- u is the flow speed (m/s)
- L is a characteristic length (m)
- ν is the kinematic viscosity of the fluid (m^2/s).

The objective is to visualize the particle distribution within the system once the lid is moved over it at a specified velocity until reaching a steady state. This experiment reveals a discernible flow pattern characterized by smaller circular currents forming at the bottom corners. We implement this in both serial and parallel fashion. For the serial implementation, it is implemented on two lattice schema of (300, 300) and (100, 100) for 20,000 timesteps. The visualizations are :

For the parallel part, we use the BWUniCluster where we are not inhibited by CPU power, and can freely test 300x300 and 500x500 with 100000 time-steps and more. We test each of those grids with an increasing number of processes, from 4 up to 900. We then calculate the million lattice updates per second (MLUPS) that it took for each of those cases. The equation is given by:

$$\text{MLUPS} = \frac{L_x \cdot L_y \cdot \text{steps}}{t}$$

where L_x and L_y represent the dimensions of the grid and t denotes the time taken. The visualizations of the few of the process is denoted here for both lattice sizes (300, 300) and (500, 500).

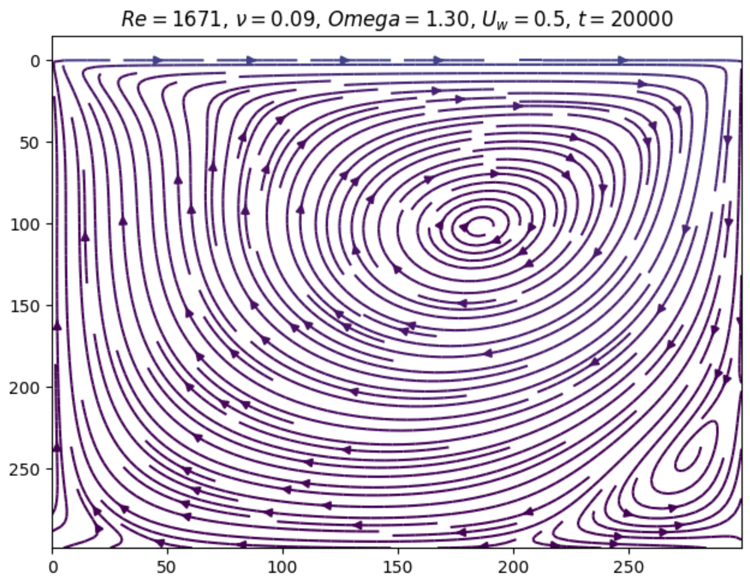

Figure 4.10: The swirls at the sides are visible for higher Reynolds number

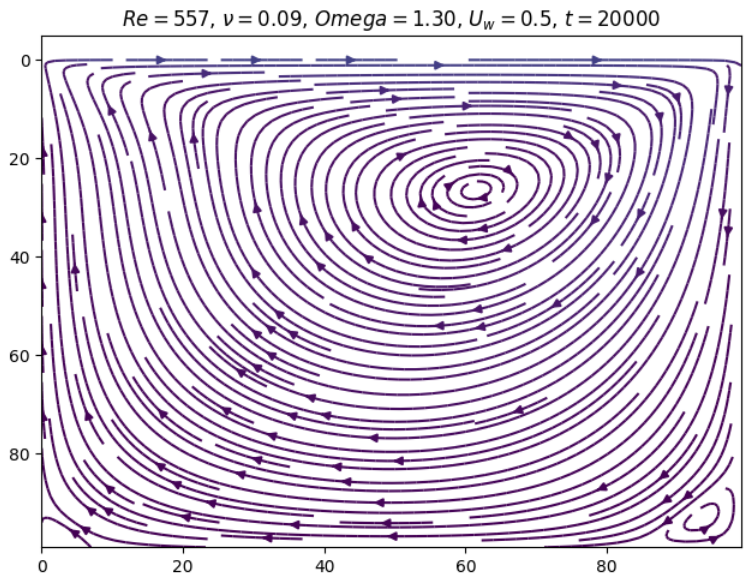

Figure 4.11: For the low Reynolds number, the side swirls are not proper

The swirls at the bottom corners are very prominent as expected and the runtime during parallelization on the cluster has been very justifiable.

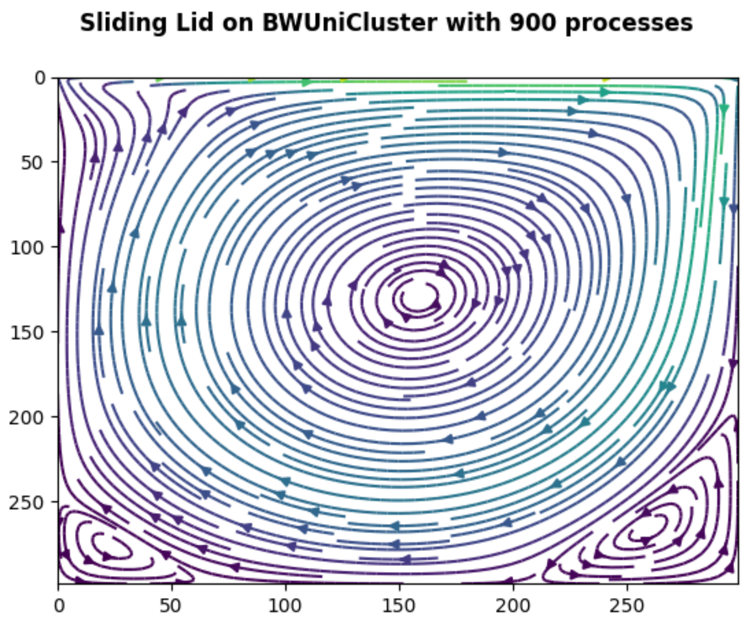

Figure 4.12: Parallel Implementation at BWUniCluster for 300 X 300 lattice

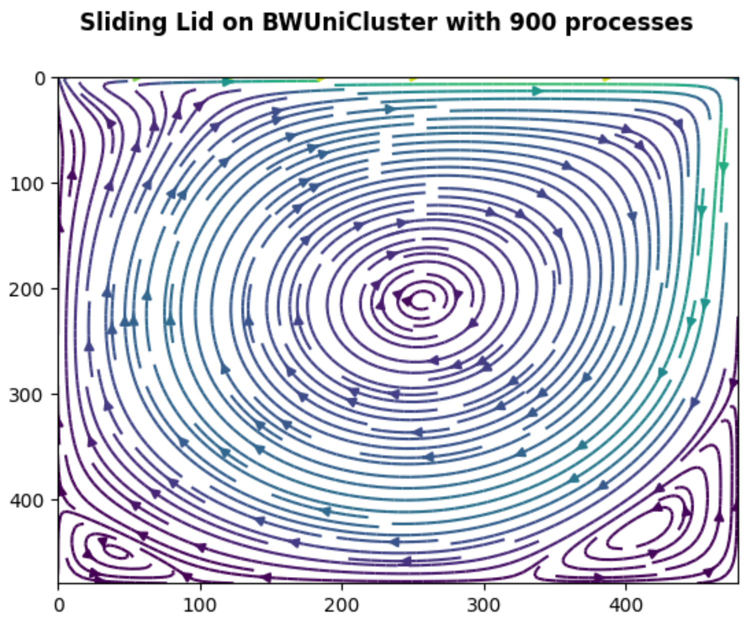

Figure 4.13: Parallel Implementation at BWUniCluster for 500 X 500 lattice

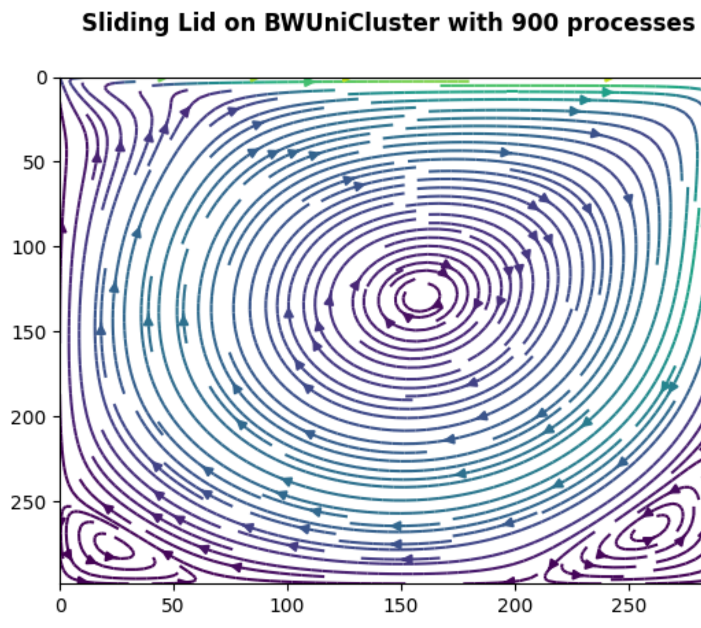

Figure 4.14: Parallel Implementation at BWUniCluster for 300 X 300 lattice

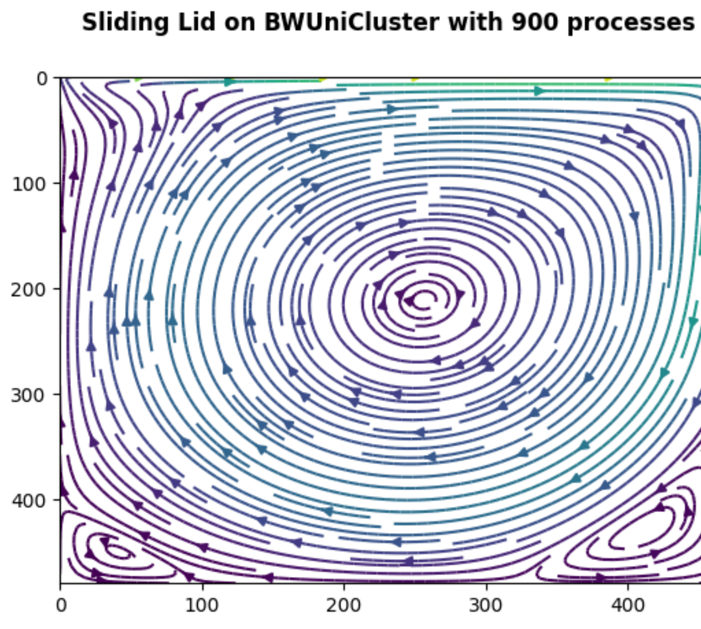

Figure 4.15: Parallel Implementation at BWUniCluster for 500 X 500 lattice

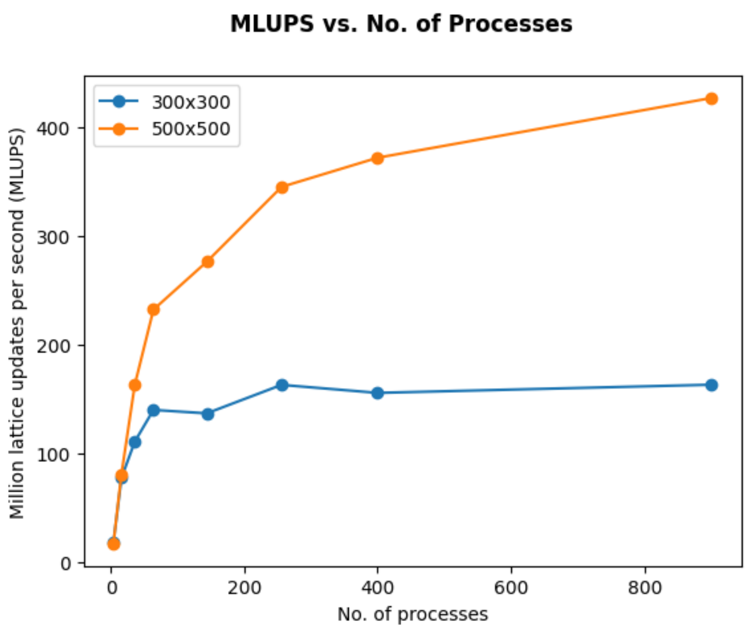

Figure 4.16: The MLUPS vs. number of processes plot for 300x300 and 500x500 grids.

4.5 Reproducibility

The code and all the visualizations along with each experiment is fully reproducible. The code is available at
`https://github.com/RK-UNI-Freiburg/HPC_SO23.git`.

5

Conclusion

This paper provides a comprehensive overview of the Lattice Boltzmann Method (LBM) and its practical applications, with a focus on parallelization using High-Performance Computing (HPC) techniques. It begins by emphasizing the pivotal role of HPC in advancing scientific and engineering fields, particularly in fluid dynamics simulations. The unique features of LBM, such as its lattice-based grid structure and versatility in handling complex geometries, are highlighted along with its broad applications in porous media flows, microfluidics, and multiphase interactions.

The theoretical foundations of LBM, including fundamental equations, discretization methods, and boundary handling techniques, are discussed in detail. The paper then delves into the systematic implementation of LBM, focusing on translating equations into Python code and employing parallelization techniques using MPI. Spatial domain decomposition and communication strategies are outlined to facilitate efficient large-scale simulations.

In Chapter 4, the paper validates the implementations through various experiments, including shear wave decay, Couette flow, Poiseuille flow, and lid-driven cavity simulations. Results from these experiments are compared with analytical solutions, demonstrating the accuracy and effectiveness of the implemented methods.

Furthermore, the paper explores parallelization using the BWUniCluster 2.0, highlighting the exponential reduction in runtime, particularly for large time-steps. The scalability of the number of processes is evaluated using the MLUPS metric, indicating optimal performance with a certain number of processes before diminishing returns occur.

In summary, the paper provides a comprehensive overview of LBM, from its theoretical underpinnings to practical implementations and performance evaluations, showcasing its efficacy in simulating fluid dynamics and its potential for large-scale parallel simulations.

Bibliography

[1] Jean Boon. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* 2003.

[2] Shiyi Chen and Gary D. Doolen. *LATTICE BOLTZMANN METHOD FOR FLUID FLOWS.* 1998.

[3] Linlin Fei, Kai H. Luo, and Qing Li. *Three-dimensional cascaded lattice Boltzmann method: Improved implementation and consistent forcing scheme.* American Physical Society, 2018.

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

[5] Lars Pastewka and Andreas Greiner. *HPC with python: An mpi-parallel implementation of the lattice boltzmann method.* Proceedings of the 5th bwHPC Symposium, 2019.

[6] Sauro Succi. *The lattice Boltzmann equation: for fluid dynamics and beyond.* Oxford university press, 2001.

[7] Zhaoxia Yang. *Analysis of Lattice Boltzmann Boundary Conditions.* PhD thesis, Universität Konstanz, 2007.

[8] Berk Hess. Determining the shear viscosity of model liquids from molecular dynamics simulations. *Journal of Chemical Physics*, 2002.

[9] E.M. Landau, L. D.; Lifshitz. *Fluid Mechanics (2nd ed.).* Elsevier, 1987.

[10] *HPC with Python Lectures at Uni-Freiburg*, 2023.

[11] Sakineh Abdi and G. Bitsuamlak. Asynchronous parallelization of a cfd solver. *Journal of Computational Engineering*, 2015.

[12] G. K Batchelor. *An Introduction to Fluid Dynamics.* Cambridge University Press, 1967.