
# High Performance Computing with Python

## Final Report

Harinarayan Raj Tilak

5050588

harinarayan.raj.tilak@students.uni-freiburg.de

October 24, 2022

# Contents

# 1

# Introduction

Fluid dynamics simulations are done to analyze and study the motions of fluids under defined conditions. They have numerous applications from improved weather forecast models [1] to better medical treatment strategies [2]. A number of simulation methods exist [3].But in this report, it is specifically discussed about and a discretized fluid model is implemented using the lattice boltzmann method. The functioning of the simulation model is represented by displaying various phenomenas that a fluid would display under different conditions. A simulation is also done to represent a fluid when present in a container that has a lid moving along it's length at constant velocity. This moving lid simulation is done for different reynolds numbers. At the end, the performance of the moving lid simulation is shown when scaled up to higher lattice points with comparison to a parallelized version of it.

# 2

# Theory

The lattice boltzmann method is primarily made up of a Probability Density Function (PDF) and the Boltzmann Transport Equation (BTE). In addition, boundary conditions along the edges of the simulation space are used to simulate specific conditions.

## 2.1 The Probability Density Function (PDF)

In order to represent a discretized model of the phase space, considering $\mathbf{r}$ to be positions, $\mathbf{v}$ to be velocities and $t$ to be the simulation time-step or iteration of the molecules being observed, The following function 2.1 is used to represent the probability of finding a certain molecule somewhere within the the phase space.

$$f(\mathbf{r}, \mathbf{v}, t) \tag{2.1}$$

This function has the density and velocity of the lattice space encoded within it. In this report, the variables $\mathbf{u}$ and $\mathbf{v}$ are used interchangeably to represent velocity.

By representing the position $\mathbf{r}$ of 2D space using the $x$ and $y$ as the two axes, the PDF notation can be rewritten to $f(x, y, \mathbf{v})$ for each time-step. Using this, the density is represented with the following equation 2.2.

$$\rho(x, y) = \int d\mathbf{v} f(x, y, \mathbf{v}) \tag{2.2}$$

The velocity field $\mathbf{v}$ is represented using the following equation.

$$\mathbf{v}(x, y) = \frac{1}{\rho(x, y)} \int d\mathbf{v} f(x, y, \mathbf{v}) \cdot \mathbf{c}(\mathbf{v}) \tag{2.3}$$

Where $\mathbf{c}$ is a function that produces the velocity set that is used to represent the direction of velocity propagation as shown in figure 2.1(a).

## 2.2 The Boltzmann Transport Equation (BTE)

In order to perform a simulation step, a differential of the PDF is required with respect to time. From this the following BTE 2.4 is arrived at.

$$\frac{\partial f(\mathbf{r}, \mathbf{v}, t)}{\partial t} + \mathbf{v} \nabla_{\mathbf{r}} f(\mathbf{r}, \mathbf{v}, t) + \mathbf{a} \nabla_{\mathbf{v}} f(\mathbf{r}, \mathbf{v}, t) = C(f) \tag{2.4}$$

The l.h.s of 2.4 is considered the "streaming" or the "propagation" step and the r.h.s the "collision" or the "relaxation" step. The streaming step just displaces the probability density in real

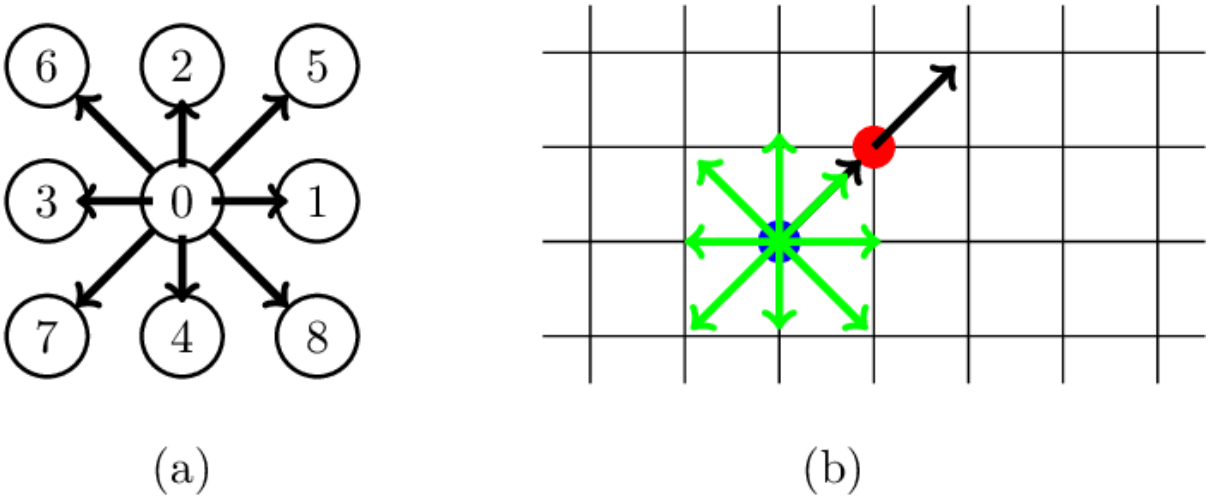


Figure 2.1: Discretization of the BTE. (a) Discretization on the velocity space according to D2Q9. (b) Uniform 2D grid for the discretization in the physical space. [4]

and the velocity space. The collision part of the equation cannot be modelled as a continuous process using a differential equation as done with the streaming part, since the time duration for a single collision process lasts for a time duration that is much lesser than that of the streaming step. Hence for the collision step, a relaxation time approximation is used called the BGK operator [5]. From this, the approximation for collision $C(f)$ at each iteration can be written using consistent notation as the following equation 2.5.

$$\left(\frac{\partial f(\mathbf{r},\mathbf{v})}{\partial t}\right)_{coll} = \frac{f^{eq}(\mathbf{r},\mathbf{v}) - f(\mathbf{r},\mathbf{v})}{\tau} \tag{2.5}$$

Using equation 2.5, the scattering effect of collision is replaced by a model that causes the function $f$ to tend towards an equilibriated state at the rate of a relaxation time constant $\tau$. the function $f^{eq}(\mathbf{r},\mathbf{v})$ is elaborated on in the implementation section. The variable $\tau$ is converted to omega ($\omega$) with the relation $\tau = 1/\omega$ and is implemented in this form thought this project and report.

## 2.3 Boundary Conditions

Boundary conditions represent operations that happen along the edge of the simulation space. The "boundary nodes" and the "interior nodes" are referred to by calling them the "dry nodes" and the "wet nodes" respectively. While performing the following operations, the boundary is simulated at points that are at half the distance between the dry nodes and the wet nodes at the edge.

### 2.3.1 Rigid Wall

Forces that get transferred onto the dry nodes after the streaming step get reflected back onto the wet nodes. This can be represented using the following equation 2.6.

$$f_j(\mathbf{x}_w, t + \Delta t) = f_i(\mathbf{x}_d, t) \tag{2.6}$$

Where $\mathbf{x}_w$ are the wet nodes, $\mathbf{x}_d$ is the dry nodes for the same boundary and the index $j$ denotes the opposite channel of the index $i$. Eg, the opposite channel of 1 is 3, 2 is 4, 5 is 7 and so on.

### 2.3.2 Moving Wall

This condition is similar to that of the rigid wall, with the addition of an extra term that changes proportional to the wall velocity $u_w$. as seen in the following equation 2.7.

$$f_j(\mathbf{x}_w, t + \Delta t) = f_i(\mathbf{x}_d, t) - 6w_i\rho_w(\mathbf{c}_i \cdot \mathbf{u}_w) \tag{2.7}$$

where $\rho_w$ is the density of the wet nodes along the boundary $w_i$ and $\mathbf{c}_i$ is a matrix with index $i$ number of columns which is stated in the implementation section of this report.

### 2.3.3 Pressure gradient

To emulate the movement of fluid along one direction in the simulation space, this concept is used. It involves modifying the density to a specific $\Delta p$ value and incorporated into the PDF at the start and the end along the axis of desired fluid movement creating a pressure gradient.

## 2.4 Analytical solutions

The analytical results are calculated in this project to validate the correctness of the simulation results as done in [6].

### 2.4.1 Viscosity

Viscosity in this project is calculated analytically from omega using the following equation 2.8.

$$\nu = \frac{1}{3}\left(\frac{1}{\omega} - \frac{1}{2}\right) \tag{2.8}$$

### 2.4.2 Shear wave decay

The analytical solution is checked for velocity with a "sine wave" along the x-axis. Considering this and complications in dealing with the Navier Stokes equations [7], only the x-axis is considered. Additionally, assumptions are made by considering the velocity perturbation to be small and pressure gradient is negligible to get the following equation 2.9.

$$\frac{\partial \mathbf{u}}{\partial t} = \nu\nabla^2\mathbf{u} \tag{2.9}$$

The equation 2.10 is used as the initial perturbation.

$$u_x(y, t = 0) = a_0 \sin\left(\frac{2\pi}{W}y\right) \tag{2.10}$$

And using equation 2.10 in 2.9 and differentiating, the following equation 2.11 is received as the analytical solution for the evolution of the velocity field along the x-axis over time.

$$u_x(y, t) = a_0 e^{-\nu(\frac{2\pi}{W})^2 t} \sin\left(\frac{2\pi}{W}y\right) \tag{2.11}$$

### 2.4.3 Poiseuille Flow

It is derived from the Navier Stokes equations [8], to get the following equation 2.12.

$$u_x(y) = -\frac{1}{2\rho\nu}y(h - y)\Delta p \tag{2.12}$$

Where $h$ is the max value of the $width$ of the channel.

## 2.5 Reynolds Number (Re)

This dimensionless quantity represents the degree to which the dynamics of a fluid system is either laminar or turbulent. The same reynolds number can be retained while increasing either the velocity or the viscosity for instance while proportionally modifying other variables to retain similar fluid flow. This quantity is used to represent the sliding lid simulation in this project. It is calculated using the following equation 2.13

$$\mathrm{Re} = \frac{Lu}{\nu}. \tag{2.13}$$

Where $L$ is the length of the system, $u$ is the velocity and $\nu$ is the kinematic viscosity of the fluid.

## 2.6 Parallelization

As it can be understood from the theoretical aspects of the Lattice Boltzmann method, running simulations with increasing order of lattice sizes causes it to also be more time consuming. To reduce the time taken for computations, a parallel computation strategy called the Message Passing Interface (MPI) is made use of, wherein the lattice space is divided into sub-domains and assigned to a "rank". Each rank is processed by a single CPU core called a "process". Communication between between each rank happens with the help of ghost nodes which receive information from the wet nodes of it's adjacent rank. This can be seen in the figure 2.2 where the wet nodes are in green and ghost nodes in grey color.

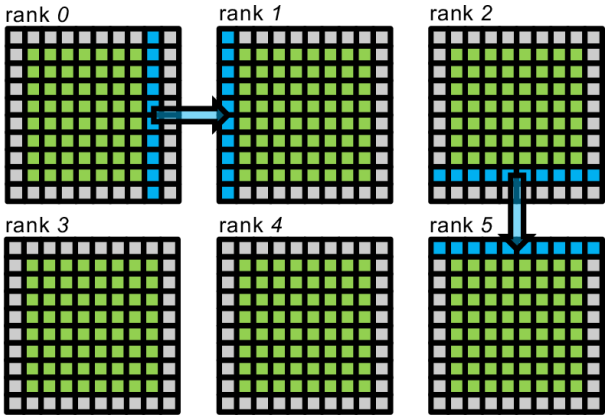


Figure 2.2: (a) Sub-domains containing ghost nodes and wet nodes divided into ranks with communication along the right and bottom directions in blue [9]

The performance of the lattice boltzmann code in this report is quantified into "Million Lattice Updates Per Second" (MLUPS), i.e. the number of grid points multiplied by number of iterations and divided by the runtime. The runtime used for this report is the wall clock time measured in seconds.

# 3

# Implementation

The simulation system is implemented in Python with the help of the numpy library for numerical calculations and Matplotlib to generate the visualizations. The shorthand "np" is used to call numpy related functions.

The following constants are declared for ease of use in the implementation. The variable names are consistent with their use in the equations stated in this report. With the index $i$ of these variables in equations representing column positions in their respective arrays.

```
w = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])
c = np.array([[0, 1, 0, -1, 0, 1, -1, -1, 1],
              [0, 0, 1, 0, -1, 1, 1, -1, -1]])
```

## 3.1 PDF representation

The discretized version of the PDF is represented in python using 9 layers of 2D matrices forming a 3D matrix as the following.

```
f = np.zeros((9,R,C))
```

where "$R$" is the width or the "y-axis" on the plots and "$C$" is the length or the "x-axis" on the plots of the simulation space. Each layer number represents the average velocity magnitude of each point in the 2D space along the direction as mentioned in figure 2.4 (a).

## 3.2 Density calculation

This operation is performed by using "np.sum()" function to implement the equation 2.2. By doing this, a 2D density array is received. This operation is implemented in the following way.

```
calc_rho = lambda f: np.sum(f, axis=0)
```

## 3.3 Velocity calculation

This operation is implemented using the equation 2.1 to receive 2 layers of 2D arrays with the 1st and the 2nd layer representing values along the y and x axes respectively. This operation is implemented in the following way.

```
calc_vel = lambda f, rho: np.dot(f.T, c[[1,0]].T).T / rho
```

## 3.4 Streaming

The "np.roll()" function is used to implement this operation. It serves the purpose of moving the fluid nodes along the directions as specified by the matrix $c$. This operation is implemented in the following way.

```
def f_stream(f):
    for i in range(1,9):
        f[i,:,:] = np.roll(f[i,:,:], c[:,i], axis=(1,0))
    return f
```

## 3.5 Collision

Using the equation 2.4, this operation is implemented in the following way.

```
def f_collision(f, omega):
    rho = calc_rho(f)
    vel = calc_vel(f, rho)
    f_eq = f_equilibrium(rho,vel)
    f = f + omega * (f_eq - f)
    return f, rho, vel, f_eq
```

Where the function "f_equilibrium(rho, vel)" is implemented in this project using the following equation.

$$f_i^{eq}(\rho, \mathbf{u}) = w_i \rho \left[1 + 3\mathbf{c}_i \cdot \mathbf{u} + \frac{9}{2}(\mathbf{c}_i \cdot \mathbf{u})^2 - \frac{3}{2}|\mathbf{u}|^2\right] \tag{3.1}$$

The implementation for the above equation 3.1 is done in the following way.

```
def f_equilibrium(rho, vel):
    var1 = np.transpose(vel, (1,0,2))
    var2 = np.transpose(np.dot(c[[1,0]].T, var1), (0,1,2))
    var3 = vel[0,:,:]**2 + vel[1,:,:]**2
    var4 = np.array([rho * w[i] for i in range(9)])
    return (var4 * (1 + 3 * var2 + 9/2 * var2**2 - 3/2 * var3))
```

## 3.6 Rigid Wall

This operation is implemented in the following way using the equation 2.6.

```
def f_rigid_wall(f, top, down, left, right):
    if top:
        f[[7,4,8],-2,:] = f[[5,2,6],-1,:]
    if down:
        f[[5,2,6],1,:] = f[[7,4,8],0,:]
    if left:
        f[[8,1,5],:,1] = f[[6,3,7],:,0]
    if right:
        f[[7,3,6],:,-2] = f[[5,1,8],:,-1]
    return f
```

## 3.7 Moving Wall

This operation is implemented in the following way using the equation 2.7

```
def f_moving_wall(f, lid_vel):
    w_rho = 2 * (f[6,-2,:] + f[2,-2,:] + f[5,-2,:]) \
                + f[3,-2,:] + f[0,-2,:] + f[1,-2,:]
    f[4, -2, :] = f[2, -2, :]
    f[8, -2, :] = f[6, -2, :] + 6 * w[6] * lid_vel * w_rho
    f[7 ,-2, :] = f[5, -2, :] - 6 * w[5] * lid_vel * w_rho
    return f
```

## 3.8 Pressure Gradient

This operation is implemented by maintaining two different density values in the following way.

```
rho_in = np.ones((width, length)) + pressure_difference/2
rho_out = np.ones((width, length)) - pressure_difference/2
```

A column of lattice points on the PDF are maintained at the beginning and end of the channel to induce this pressure gradient effect on the channel. These channels are modified at the end of each iteration in the following way.

```
    f[:,1:-1, 0] = f_equilibrium(rho_in, vel)[:,1:-1,-2] \
                    + (f[:,1:-1,-2] - f_eq[:,1:-1,-2])
    f[:,1:-1,-1] = f_equilibrium(rho_out, vel)[:,1:-1, 1] \
                    + (f[:,1:-1, 1] - f_eq[:,1:-1, 1])
```

## 3.9 Parallelization

Parallelization of the sliding lid simulation is implemented using the "mpi4py" library. The functions implemented by this library namely *Shift*, *Sendrecv* and *Allreduce* are used to perform this perform this operation. They perform the operation of identifying the ranks relative to the current rank, sending data across ranks and collecting all data all ranks to a single rank respectively.

The "velocity storage function" uses the *Allreduce* function. This function was provided by the lecture material [4] to store snapshots of the velocities. The "communicator function" uses the *Sendrecv* and *Allreduce* functions.
First a communicator is created in the Cartesian space. using the following line.

```
comm = MPI.COMM_WORLD
CommCart = comm.Create_cart((y_decomp, x_decomp), periods=(False, False),
                                        reorder=False)
```

Where the *x_decomp* and *y_decomp* represent the number of sub-sections the Cartesian space is divided along the x and y axis respectively, *periods* represents the possibility of communicating across boundaries of the simulation space and *reorder* represents the possibility of reordering ranks. Then the "communicator function" is created the following way.

```
def f_communicator(f, commCart):
    top_src, top_dst = commCart.Shift(0, -1)
    bot_src, bot_dst = commCart.Shift(0, +1)
    lef_src, lef_dst = commCart.Shift(1, -1)
    rig_src, rig_dst = commCart.Shift(1, +1)
    p1 = f[:,  1,  :].copy()
    p2 = f[:, -1,  :].copy()
    commCart.Sendrecv(p1, top_dst, recvbuf=p2, source=top_src)
    f[:, -1,  :] = p2
    p1 = f[:, -2,  :].copy()
    p2 = f[:,  0,  :].copy()
    commCart.Sendrecv(p1, bot_dst, recvbuf=p2, source=bot_src)
    f[:,  0,  :] = p2
    p1 = f[:,  :,  1].copy()
    p2 = f[:,  :, -1].copy()
    commCart.Sendrecv(p1, lef_dst, recvbuf=p2, source=lef_src)
    f[:,  :, -1] = p2
    p1 = f[:,  :, -2].copy()
    p2 = f[:,  :,  0].copy()
    commCart.Sendrecv(p1, rig_dst, recvbuf=p2, source=rig_src)
    f[:,  :,  0] = p2
    return f
```

# 4

# Results

The notation $[L, W]$ is used to represent the shape of the simulation space where "$L$" and "$W$" are the *length* and *width* respectively but without taking dry nodes or ghost nodes into consideration. The same notation is used to represent number of processes along an axis during parallel operation. For example: [3,2] processes would mean that the lattice is evenly divided into 3 sub-parts along the $x$-axis, 2 times along the $y$-axis. The plots contain no ghost nodes or dry nodes. The values for $x$ and $y$ range from $(0, L)$ and $(0, W)$ respectively.

## 4.1 Shear Wave Decay

### 4.1.1 Density Variation

After choosing an initial distribution of $\rho(\mathbf{r}, t) = 0.5 + 0.01 \sin\left(\frac{2\pi x}{L}\right)$ with and $\mathbf{u}(\mathbf{r}, t) = 0$ at $t = 0$, the following density evolution is produced as shown in the following figures 4.1 and 4.2.

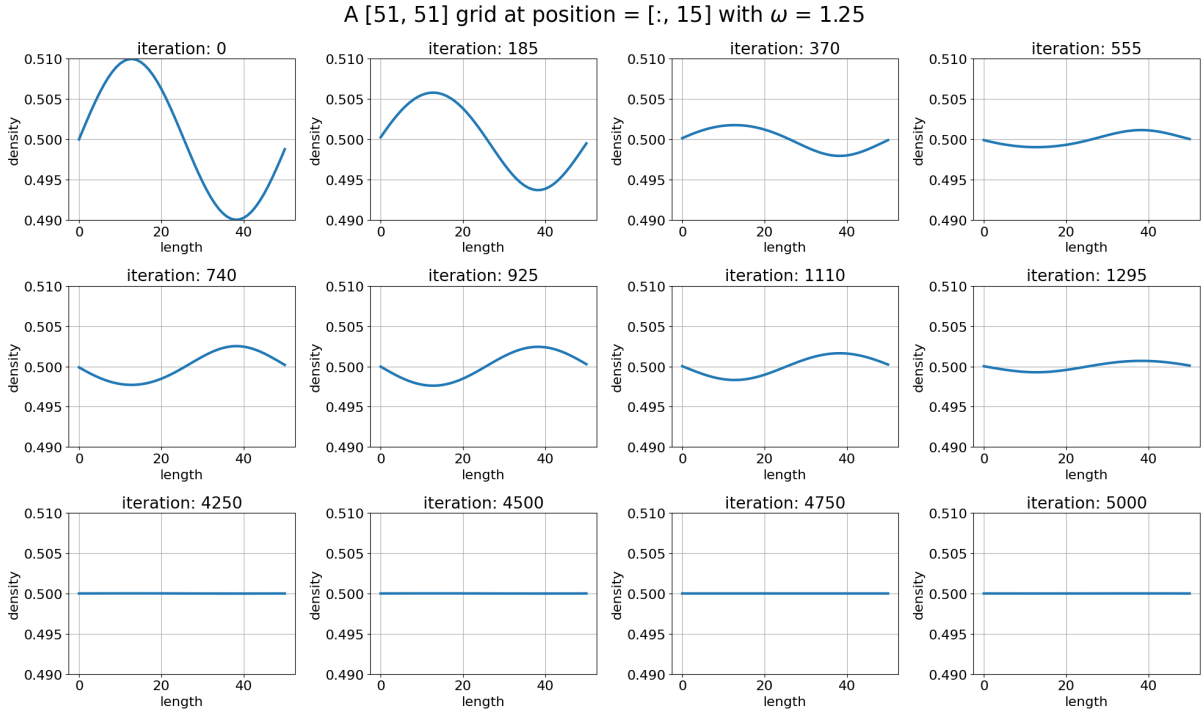


Figure 4.1: Plot: Density of fluid after density variation

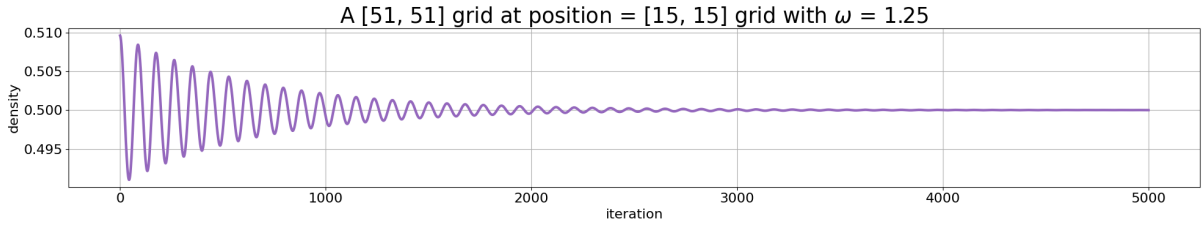


Figure 4.2: Plot: Density evolution over time period after density variation

### 4.1.2 Velocity Variation

After choosing an initial distribution of $\rho(\mathbf{r}, 0) = 1$ and $u_x(\mathbf{r}, 0) = 0.01 \sin\left(\frac{2\pi y}{W}\right)$, the following density evolution is produced as shown in the following figures 4.3 and 4.4.

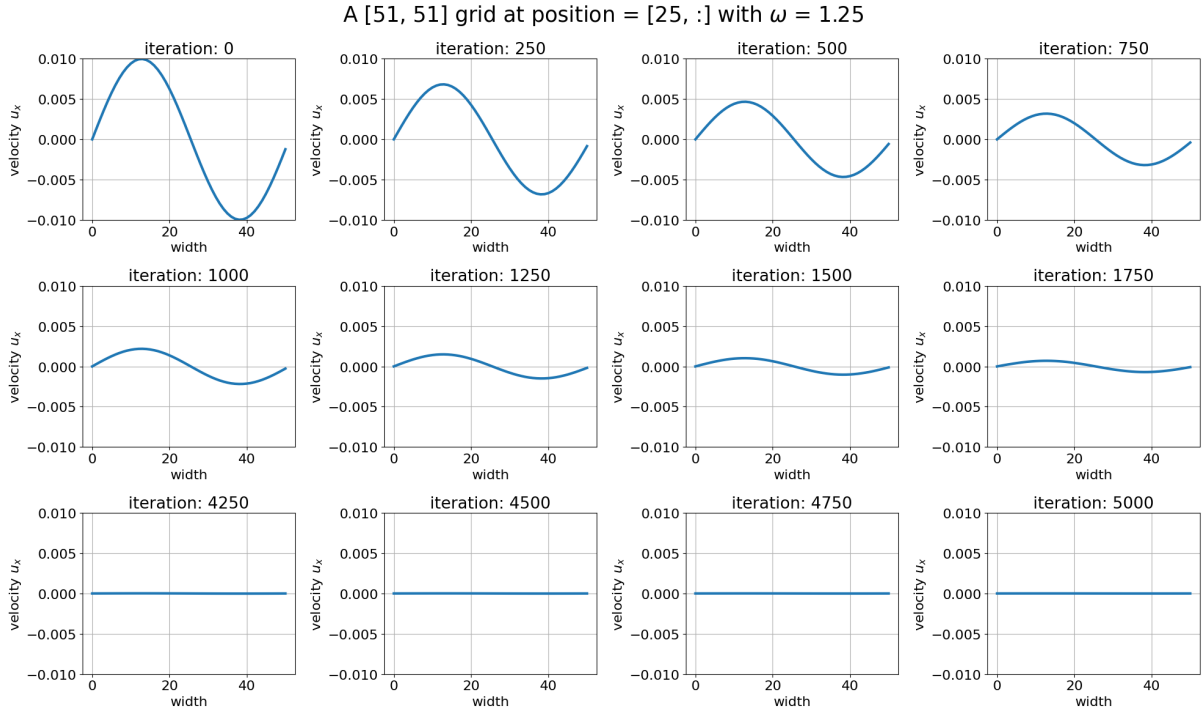


Figure 4.3: Plot: Velocity of fluid after velocity variation

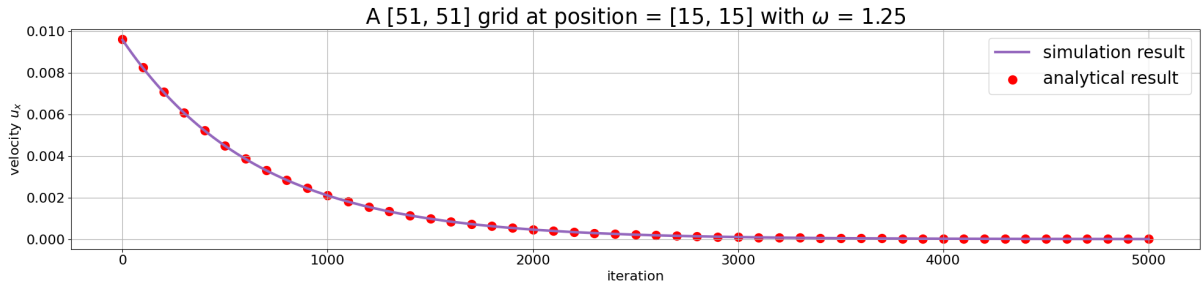


Figure 4.4: Plot: Velocity evolution over time period after velocity variation

The simulation viscosity matches the analytical viscosity result only for higher values of omega as shown in the following figure 4.5.

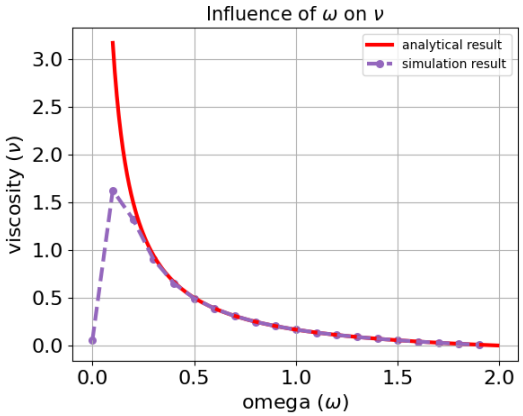


Figure 4.5: Plot: Simulation results diverging from analytical results for low values of omega with velocity variation

## 4.2 Couette Flow

After choosing an an initial distribution of $\rho(\mathbf{r}, 0) = 1$ and $\mathbf{u}(\mathbf{r}, 0) = 0$, applying rigid wall boundary at the bottom and moving wall boundary at the top, the following results are obtained as shown in the following figure 4.6.

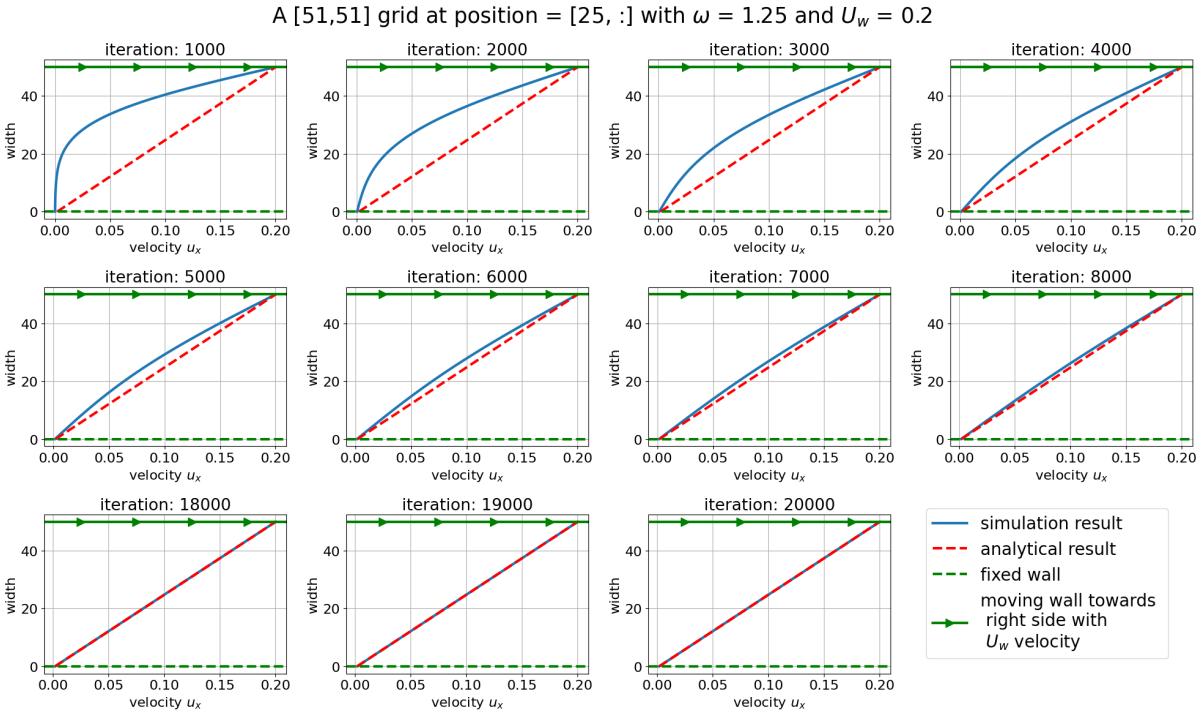


Figure 4.6: Plot: Couette flow

## 4.3 Poiseuille Flow

After choosing an an initial distribution of $\rho(\mathbf{r}, 0) = 1$, $\mathbf{u}(\mathbf{r}, 0) = 0$ and applying rigid wall boundary for the top and bottom, The results are produced as shown in the following figures 4.7 and 4.8.

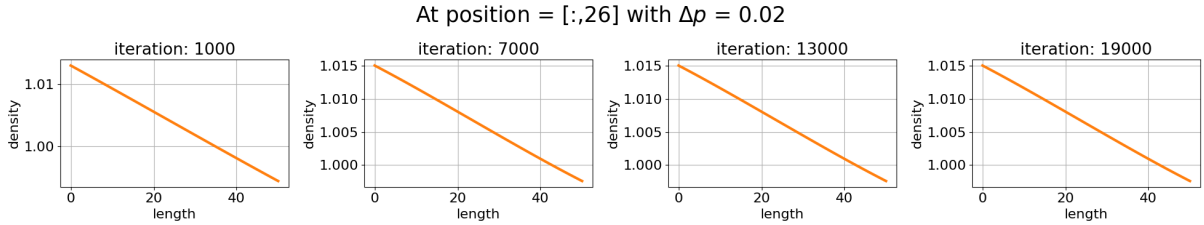


Figure 4.7: Plot: Pressure gradient in poiseuille flow

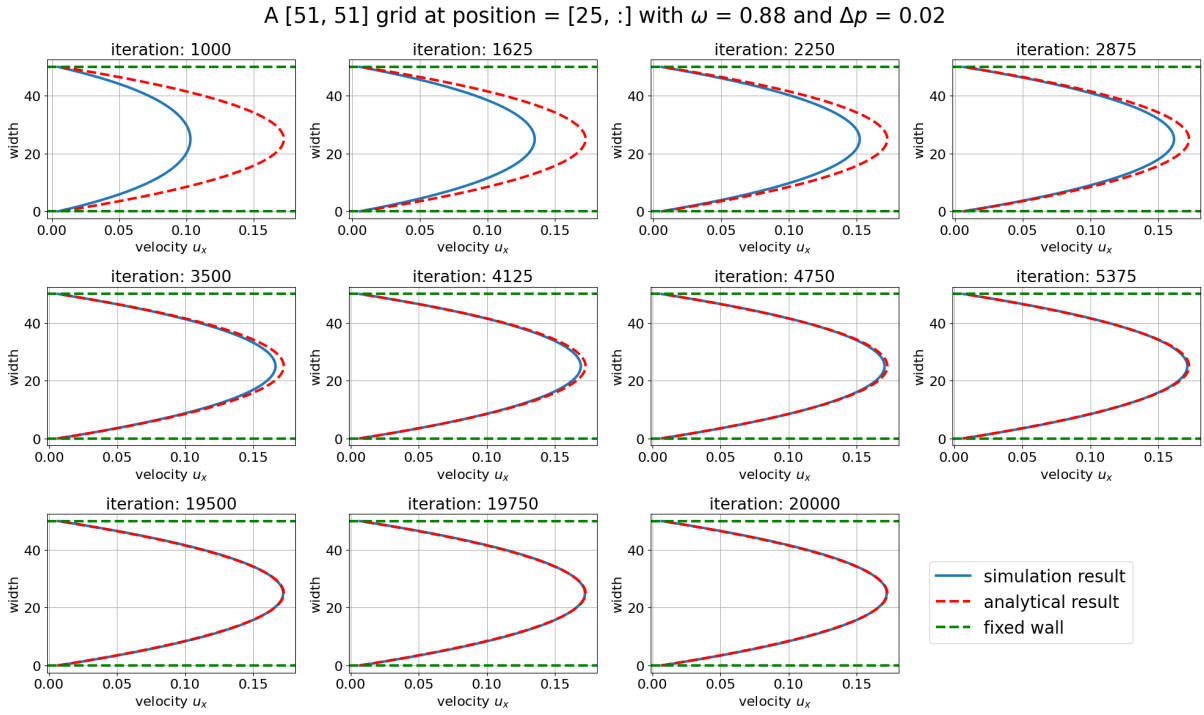


Figure 4.8: Plot: Poiseuille flow

## 4.4 Sliding Lid

After choosing an initial distribution of $\rho(\mathbf{r}, 0) = 1$, $\mathbf{u_x}(\mathbf{r}, 0) = 0$ and $\mathbf{u_y}(\mathbf{r}, 0) = 0$, the results produced are shown in the following figures 4.9 and 4.10.

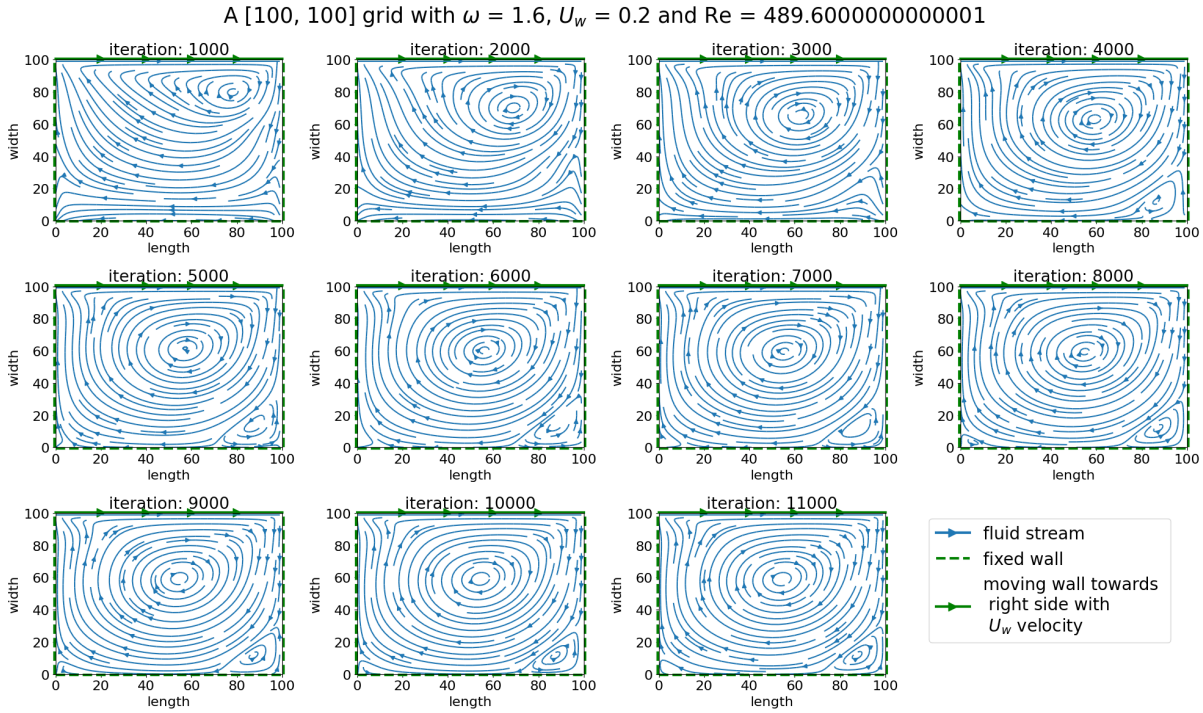


Figure 4.9: Plot: Serial sliding lid simulation

## 4.5 Parallelization

After parallelization, the results produced are shown in the following figure 4.10.

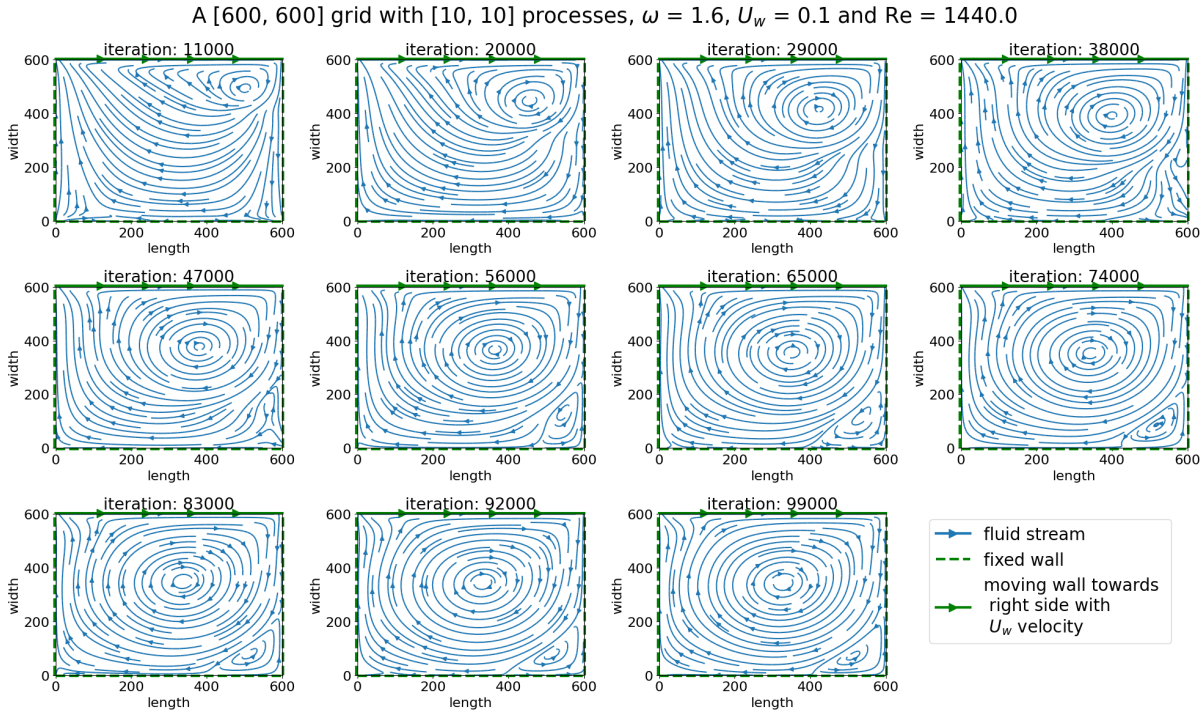


Figure 4.10: Plot: Parallelized sliding lid simulation

From figure 4.11, the performance gain from parallelization can be seen for different grid sizes.

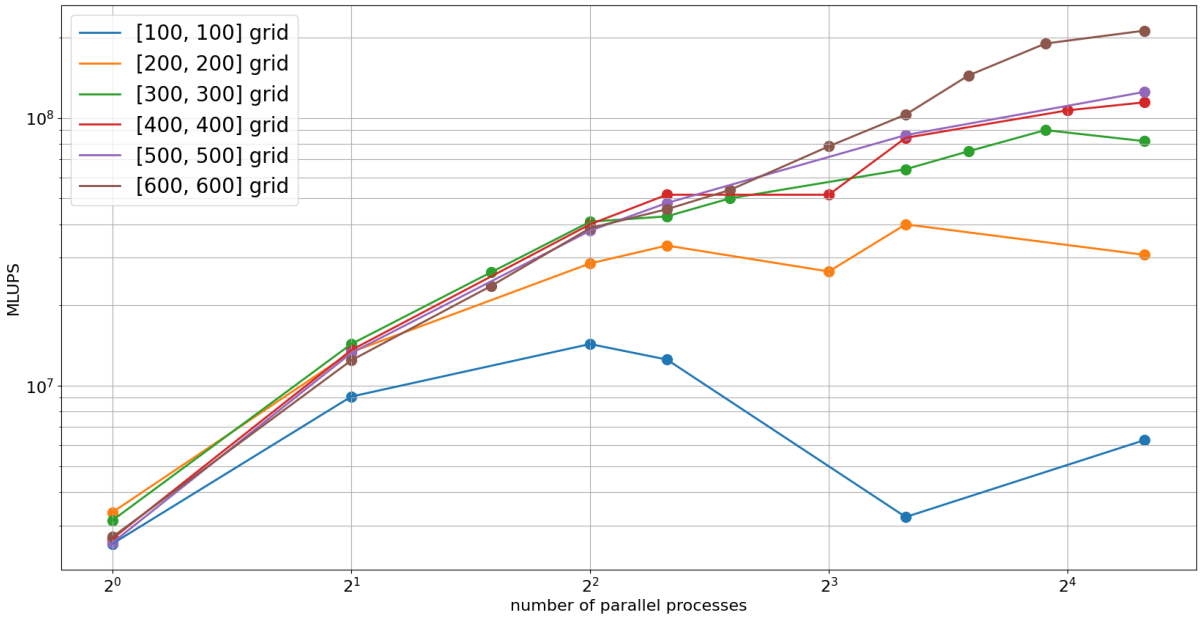


Figure 4.11: Plot: Comparison of performance gain for different grid sizes from parallelized sliding lid simulation

# 5

# Conclusion

By reading the contents of this paper, one can understand the theoretical and implementation aspects for a simple fluid dynamics simulation with the lattice boltzmann method. A demonstration of the the properties of fluids namely shear wave decay, velocity decay, couette flow and poiseuille flow was done. The theoretical aspects required to implement these properties were elaborated on. The simulation of a fluid present in a closed container having a sliding lid was implemented to demonstrate the overall working of this fluid simulation method. Towards the end, the sliding lid demonstration was parallellized using MPI and the performance gain through this parallelization was represented.

# Bibliography


[1] Numerical weather prediction. https://en.wikipedia.org/wiki/Numerical_weather_prediction. Accessed: 2022-10-24.

[2] Luke Reid. An introduction to biomedical computational fluid dynamics. In *Advances in Experimental Medicine and Biology*, pages 205–222. Springer International Publishing, 2021.

[3] Computational fluid dynamics. https://en.wikipedia.org/wiki/Computational_fluid_dynamics. Accessed: 2022-10-24.

[4] Lars Pastewka and Andreas Greiner. Lecture notes of "high-performance computing: Fluid mechanics with python", 2022.

[5] Bhatnagar–gross–krook operator. https://en.wikipedia.org/wiki/Bhatnagar%E2%80%93Gross%E2%80%93Krook_operator. Accessed: 2022-10-24.

[6] Linlin Fei, Kai H. Luo, and Qing Li. Three-dimensional cascaded lattice boltzmann method: Improved implementation and consistent forcing scheme. *Phys. Rev. E*, 97:053309, May 2018.

[7] Navier stokes existence and smoothness. https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_existence_and_smoothness. Accessed: 2022-10-24.

[8] Hagen–poiseuille equation. https://en.wikipedia.org/wiki/Hagen%E2%80%93Poiseuille_equation#Plane_Poiseuille_flow. Accessed: 2022-10-24.

[9] Lars Pastewka and Andreas Greiner. Hpc with python: An mpi-parallel implementation of the lattice boltzmann method, 2019.