universität freiburg

High Performance Computing: Fluid Mechanics with Python

Arkadyuti Kundu

5454879
University of Freiburg

Professors & Instructors: **Andreas Greiner, Andrea Codrignani, Lars Pastewka**

14th August, 2023

Contents

Chapter 1

Introduction

The advent of High-Performance Computing (HPC) has revolutionised various scientific and engineering fields, enabling the simulation of complex phenomena with unprecedented accuracy and efficiency. One such powerful technique gaining traction in the realm of computational fluid dynamics is the Lattice Boltzmann Method (LBM). This report explores the implementation of LBM using the Message Passing Interface (MPI) to achieve parallelization, thereby harnessing the full potential of HPC resources.

The Lattice Boltzmann Method is a versatile numerical approach used to model fluid flows and complex fluid-structure interactions. Unlike traditional Navier-Stokes solvers, LBM relies on a lattice-based grid and discrete velocity distribution functions, making it particularly suitable for parallelization [1]. The method is based on the Boltzmann equation and offers advantages in handling intricate geometries and boundary conditions, making it a valuable tool in simulating fluid dynamics in porous media, microfluidics, and multiphase flows, among other applications.

LBM has found applications in a wide range of scientific and engineering fields. One notable application is in simulating fluid flows through porous media, which has significant implications in petroleum engineering, groundwater modelling, and environmental studies [2]. Additionally, LBM has been extensively used in studying complex fluid dynamics at the microscale, such as blood flow in capillaries and the behaviour of suspensions in microchannels [3]. Its ability to model multiphase flows also makes it invaluable in simulating phenomena like droplet dynamics, bubble formation, and liquid-gas interactions [4].

The Lattice Boltzmann Method offers several advantages over traditional numerical methods. Firstly, its lattice structure enables straightforward handling of complex geometries, obviating the need for complex mesh generation [5]. Secondly, LBM naturally incorporates microscopic interactions, making it well-suited for simulating mesoscopic and microscale phenomena. Thirdly, the method is intrinsically parallelizable, facilitating its efficient execution on HPC systems.

Given the computational demands associated with LBM simulations, HPC systems offer a natural solution to expedite computations and tackle larger, more complex problems. Parallelizing LBM using MPI enables efficient distribution of computational tasks across multiple processors or nodes, thereby reducing the overall simulation time [6]. Moreover, HPC resources allow researchers to explore finer spatial and temporal resolutions, leading to more accurate and detailed results.

Chapter 2

Methods

2.1 Lattice Boltzmann Method: Equations and Boundary Handling

The Lattice Boltzmann Method (LBM) is a powerful numerical technique used to simulate fluid flows and complex phenomena in various scientific and engineering applications. In this section, we explore the fundamental equations underlying LBM, including the continuous and discretized Boltzmann Transport Equation (BTE), the equilibrium distribution, the streaming and collision terms, and boundary handling techniques.

2.1.1 Continuous Boltzmann Transport Equation (BTE)

The Boltzmann Transport Equation (BTE) formulates the time evolution of the particle probability density function $f(\mathbf{r}, \mathbf{v}, t)$, where $\mathbf{v}$ represents the microscopic velocity, $\mathbf{r}$ denotes the position of particles, and t is time. The BTE describes the behaviour of particles in a gas or fluid and is given by [7]:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{r}} f + \mathbf{a} \cdot \nabla_{\mathbf{v}} f = \mathbf{C}(f) \tag{2.1.1}$$

where $\frac{\partial f}{\partial t}$ represents the time derivative, $\mathbf{v} \cdot \nabla_{\mathbf{r}} f$ and $\mathbf{a} \cdot \nabla_{\mathbf{v}} f$ account for the advection terms in space and velocity, respectively, and $\mathbf{C}(f)$ is the collision operator. The collision operator models the interactions between particles, leading to the relaxation of the distribution function towards an equilibrium state.

2.1.2 Discretization of the BTE

To make the BTE computationally feasible, we need to discretize it in both space and velocity. In the LBM, the domain is divided into a lattice with nodes, and the particle velocities are defined on the lattice points. For a two-dimensional lattice (D2Q9), there are nine discrete velocities given by:

$$\mathbf{c} = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix}^T \tag{2.1.2}$$

The continuous BTE is then discretized using finite differences to obtain the following equation for each lattice node:

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau} \left[f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t) \right] \tag{2.1.3}$$

where $f_i(\mathbf{x}, t)$ is the particle distribution function at velocity $\mathbf{c}_i$, Δt is the time step, τ is the relaxation time, and f_i^{eq} is the equilibrium distribution function at $\mathbf{c}_i$. The equilibrium distribution function is typically derived using the Chapman-Enskog expansion and depends on macroscopic quantities like density and velocity.

2.1.3 Streaming and Collision Terms

The LBM simulation proceeds through two main steps at each time step: the streaming and collision steps. In the streaming step, particles propagate along the lattice velocities to neighbouring nodes, and in the collision step, the particle distribution functions relax towards their equilibrium values. This process can be represented mathematically as:

$$\underbrace{f_i(\mathbf{x} + \mathbf{c}_i\Delta t, t + \Delta t) - f_i(\mathbf{x}, t)}_{\text{streaming}} = \underbrace{-\frac{1}{\tau}\left[f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)\right]}_{\text{collision}} \tag{2.1.4}$$

2.1.4 Moment Updates

The macroscopic quantities, such as density ρ and velocity $\mathbf{u}$, are derived from the particle distribution functions. The moment updates for D2Q9 are given by [8]:

$$\rho(\mathbf{x}, t) = \sum_{i=1}^{9} f_i(\mathbf{x}, t) \tag{2.1.5}$$

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{\rho(\mathbf{x}, t)} \sum_{i=1}^{9} \mathbf{c}_i f_i(\mathbf{x}, t) \tag{2.1.6}$$

where $f_i(\mathbf{x}, t)$ is the particle distribution function at velocity $\mathbf{c}_i$.

2.1.5 Equilibrium Distribution

The equilibrium distribution function f_i^{eq} is computed using the macroscopic density ρ and velocity $\mathbf{u}$ at each lattice node. For D2Q9, the equilibrium distribution function is given by [9]:

$$\begin{aligned} f_i^{eq}(\mathbf{x}, t) = w_i\rho(\mathbf{x}, t) \\ \times [1 + 3\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x}, t) \\ + \frac{9}{2}(\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x}, t))^2 - \frac{3}{2}\mathbf{u}(\mathbf{x}, t) \cdot \mathbf{u}(\mathbf{x}, t)\Big] \end{aligned} \tag{2.1.7}$$

where w_i is the weight corresponding to the velocity $\mathbf{c}_i$ and depends on the lattice type. For D2Q9, $w_i = \frac{4}{9}$ for the central direction, $w_i = \frac{1}{9}$ for the horizontal and vertical directions, and $w_i = \frac{1}{36}$ for the diagonal directions. Index i corresponds to Figure 2.1.

2.1.6 Relaxation Parameter τ

The relaxation time τ is a crucial parameter in LBM that controls the viscosity and determines the numerical stability of the simulation. It determines how quickly the fluid converges towards equilibrium. A higher value of τ leads to slower convergence, and vice versa. The relaxation time is related to the kinematic viscosity ν through the relation [10] $\nu = \frac{1}{3}(\tau - \frac{1}{2})$.

2.1.7 Boundary Handling

Handling boundary conditions is essential to ensure accurate representation of physical scenarios in numerical simulations. There are mainly two different approaching for boundary handling, namely:

1. **Wet Nodes:** Wet nodes are lattice nodes where the boundaries are located directly on the lattice points. In other words, the boundaries coincide with the lattice points of the fluid grid.
2. **Dry Nodes:** Dry nodes are lattice nodes where the boundaries are located between the lattice points. In this approach, the boundaries are placed on the links connecting adjacent lattice nodes.

For easier implementation we will be using dry nodes here.

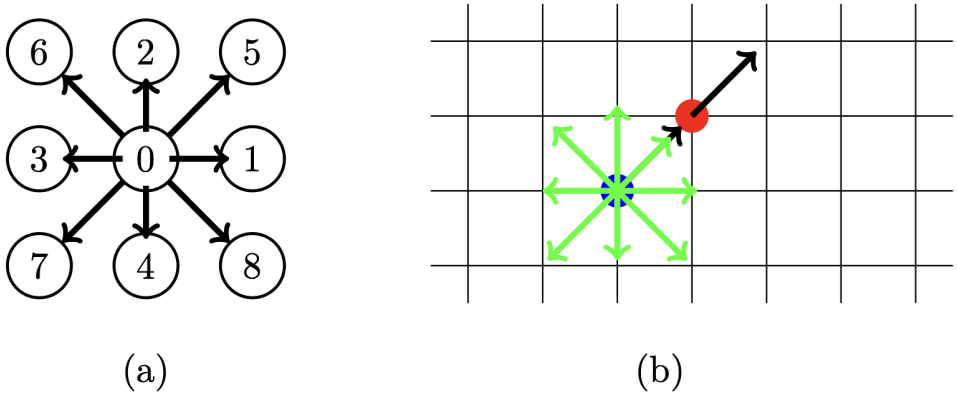

Figure 2.1: Discretization of the BTE. (a) Discretization on the velocity space according to D2Q9. (b) Uniform 2D grid for the discretization in the physical space[1].

2.1.8 Bounce-Back

In LBM, the bounce-back boundary condition is commonly used to simulate rigid walls. It enforces a no-slip condition at the boundary, meaning that the fluid particles adhere to the boundary's velocity. When a particle collides with the boundary, its velocity is reversed (bounce-back), effectively simulating reflection from the wall. If the boundary is moving with a velocity U_w, the equation is modified to consider the variation in the moments of particles [11]:

$$f_i(x_b, t + \Delta t) = f_i^\star(x_b, t) - \frac{2 w_i \rho_w c_i \cdot U_w}{{c_s}^2} \tag{2.1.8}$$

where x_b is the boundary node, $f_i^\star$ is the particle distribution function after the streaming step, ρ_w is the density at the wall, and c_i is the velocity vector corresponding to the lattice direction i.

2.1.9 Periodic Boundary Conditions (PBC) with Pressure Variation

Periodic boundary conditions ensure continuous flow by allowing fluid particles exiting one boundary to re-enter from the opposite boundary. To implement periodic boundary conditions with pressure variation, we consider the pressure difference Δp between the inlet and outlet boundaries. The density ρ at the inlet and outlet can be computed as follows [11]:

$$\rho(x = 0, t) = \rho_{out} + \Delta p, \quad \rho(x = (X - 1)\Delta x, t) = \rho_{out}$$

where X is the number of lattice nodes in the x-direction, Δx is the lattice spacing, and ρ_{out} is the constant pressure at the outlet boundary. Then, the pre-streaming distribution function $f_i^\star$ at the inlet and outlet is updated as follows:

$$\begin{aligned} f_i^\star(-\Delta x, y, t) = & f_i^{eq}(\rho_{\text{in}}, \mathbf{u}(x = (X - 1)\Delta x, y, t)) \\ & + (f_i^\star(x = (X - 1)\Delta x, y, t) \\ & - f_i^{eq}(\rho_{\text{out}}, \mathbf{u}(x = (X - 1)\Delta x, y, t))), \quad \text{for inlet} \end{aligned} \tag{2.1.9}$$

$$\begin{aligned} f_i^\star(X\Delta x, y, t) = & f_i^{eq}(\rho_{out}, \mathbf{u}(x = 0, y, t)) \\ & + (f_i^\star(x = 0, y, t) \\ & - f_i^{eq}(\rho_{out}, \mathbf{u}(x = 0, y, t))), \quad \text{for outlet} \end{aligned} \tag{2.1.10}$$

where y is the coordinate in the y-direction, f_i^{eq} is the equilibrium distribution function, and $\mathbf{u}$ is the macroscopic velocity at each lattice node.

Note: The periodic boundary conditions are performed before the streaming step, unlike the bounce-back boundary conditions which are performed after the streaming step.

Chapter 3

Implementation

Here, we will explore the step-by-step implementation of LBM and the parallelization techniques employed to enhance computational efficiency. The focus lies on how the fundamental equations of LBM are translated into Python code to simulate fluid dynamics accurately. The entire implementation assumes a discretization of the physical domain using the D2Q9 lattice, with the horizontal axis denoted as 'x' and the vertical axis as 'y'. Throughout the codes, we make use of Numpy[1] and mpi4py[2] libraries.

3.1 Serial Routine

The LBM simulation consists of several steps, such as the streaming step, collision step, and boundary condition application.

Algorithm 1 Lattice Boltzmann Method Main Routine

procedure LBM
    Initialize LBM parameters ($x, y,$ density, velocity, omega, $\epsilon,$ wall_velocity, $rho_{\text{in}}, rho_{\text{out}},$ process_info, communicator)
    for step $\leftarrow 0$ **to** *total_steps* **do**
        if step $= 0$ **then**
            Compute the equilibrium distribution function (pdf) using the initial density and velocity
        end if
        if periodic_boundary_conditions **then**
            Apply periodic boundary conditions with pressure variations
        end if
        Perform the streaming step to move particles to neighbouring grid points
        if boundary_conditions **then**
            Apply the specified boundary conditions
        end if
        Compute the density and velocity from the updated pdf
        Perform the collision step to relax the probability density towards equilibrium
    end for
end procedure

Here is a brief explanation of the variables used in the code:

1. x and y: The size of the grid in the x and y dimensions, respectively.
2. `density`: An array to store the density values with shape (x, y).

[1] https://numpy.org
[2] https://mpi4py.readthedocs.io/en/stable/

3. `velocity`: An array to store the velocity components (u and v) at each grid point with shape (2, x, y).

4. `pdf`: The grid of probability density functions (pdf) for each velocity direction with shape (9, x, y).

5. `oldPdf`: Probability density function at the earlier step, i.e., before streaming.

6. c: The velocity set containing the 9 discrete velocities in 2D (D2Q9).
   $c = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix}$.

7. ω: The relaxation factor (ω) used in the collision step.

8. `weights`: The weights corresponding to each velocity direction.
   `weights` = $\left[\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right]$.

9. ϵ: Dissipation rate.

10. `wall_velocity`: The velocity of the moving wall (for boundary conditions).

11. ρ_{in} and ρ_{out}: The density values at the inlet and outlet, respectively (for boundary conditions).

12. `process_info`: Information about the MPI process (for parallelization).

13. `communicator`: MPI communicator (for parallelization).

14. `s_lid`: Whether to perform Sliding Lid simulation.

Explanation of the different algorithms used and code flow:

Algorithm 1 enters a loop that runs for the specified number of total steps. In each step, Algorithm 1 performs the following operations:

1. If it is the first step (`step = 0`), the algorithm computes the equilibrium distribution function (pdf) using the initial density and velocity of the fluid. [Algorithm 2]

Algorithm 2 Pseudocode for Equilibrium Distribution

procedure ComputeEquilibriumDistribution(density, velocity)
  velocity_squared $\leftarrow$ np.einsum($'axy, axy- > xy'$, velocity, velocity)
  equilibrium $\leftarrow$ np.einsum($'i, xy- > ixy'$, weights, density)$\times$
    $(1 + 3 \times$ np.einsum($'ai, ixy- > axy'$, c.T, velocity)$+$
    $(9/2) \times$ np.einsum($'ai, ixy- > axy'$, c.T, velocity$)^2 - (3/2)\times$
    velocity_squared)
  return equilibrium ▷ Return the equilibrium distribution
end procedure

2. The algorithm then checks if periodic boundary conditions are to be applied. If so, it applies periodic boundary conditions with pressure variations. [Algorithm 3]

3. Next, the algorithm performs the streaming step, which moves particles to neighbouring grid points based on their velocities. This operation updates the probability density function (pdf) for each direction in the velocity set. [Algorithm 4]

4. If boundary conditions are specified, the algorithm applies the corresponding boundary conditions. This step ensures that the fluid behaviour near boundaries is appropriately accounted for. [Algorithm 5]

Algorithm 3 Periodic Boundary Conditions with Pressure Variations

procedure PERIODICBOUNDARYWITHPRESSURE($\rho_{\text{in}}, \rho_{\text{out}}$)

function EQDISTRIBUTION(density, velocity)

velocity_squared $\leftarrow$ np.einsum("$ax, ax \rightarrow xy$", velocity, velocity)

eq $\leftarrow$ (weights $\times$ density) $\times \Big(1 + 3 \times$ np.einsum("$ai, ixy \rightarrow axy$", c^T, velocity) $+ \frac{9}{2} \times$ np.einsum("$ai, ixy \rightarrow axy$", c^T, velocity)$^2 - \frac{3}{2} \times$ velocity_squared$\Big)$

return eq

end function

density $\leftarrow$ compute_density() ▷ Compute the density from the distribution functions

velocity $\leftarrow$ compute_velocity() ▷ Compute the velocity from the distribution functions

equilibrium $\leftarrow$ Equilibrium_Distribution(density, velocity) ▷ Compute the equilibrium distribution

equilibrium_in $\leftarrow$ EqDistribution(ρ_{in}, velocity[:, -2, :]) ▷ Compute the equilibrium distribution at the inlet

pdf[:, 0, :] $\leftarrow$ equilibrium_in + (pdf[:, -2, :] $-$ equilibrium[:, -2, :]) ▷ Update pdf at the inlet

equilibrium_out $\leftarrow$ EqDistribution(ρ_{out}, velocity[:, 1, :]) ▷ Compute the equilibrium distribution at the outlet

pdf[:, -1, :] $\leftarrow$ equilibrium_out + (pdf[:, 1, :] $-$ equilibrium[:, 1, :]) ▷ Update pdf at the outlet

end procedure

Algorithm 4 Streaming Step in LBM

procedure STREAMING

for $i \leftarrow 0$ **to** 8 **do** ▷ Loop through all discrete velocities

Shift the probability density *pdf* in the x and y directions using `np.roll` according to $\mathbf{c}_i$

end for

end procedure

Algorithm 5 Apply Boundary Conditions

procedure BOUNDARYCONDITIONS
  if s_lid **then**
    ▷ For left rigid wall
    pdf[1, 0, :] $\leftarrow$ $oldPdf[3, 0, :]$ ▷ Bounce back from left side
    pdf[5, 0, :] $\leftarrow$ $oldPdf[7, 0, :]$ ▷ Bounce back from left side
    pdf[8, 0, :] $\leftarrow$ $oldPdf[6, 0, :]$ ▷ Bounce back from left side
    ▷ For right rigid wall
    pdf[3, -1, :] $\leftarrow$ $oldPdf[1, -1, :]$ ▷ Bounce back from right side
    pdf[6, -1, :] $\leftarrow$ $oldPdf[8, -1, :]$ ▷ Bounce back from right side
    pdf[7, -1, :] $\leftarrow$ $oldPdf[5, -1, :]$ ▷ Bounce back from right side
  end if
  ▷ For bottom rigid wall
  pdf[2, :, 0] $\leftarrow$ $oldPdf[4, :, 0]$ ▷ Bounce back from bottom
  pdf[5, :, 0] $\leftarrow$ $oldPdf[7, :, 0]$ ▷ Bounce back from bottom
  pdf[6, :, 0] $\leftarrow$ $oldPdf[8, :, 0]$ ▷ Bounce back from bottom

  ▷ For top moving wall
  $\rho\_wall \leftarrow 2.0 \times (oldPdf[2, :, -1] + oldPdf[5, :, -1]$
    $+oldPdf[6, :, -1] + oldPdf[0, :, -1] + oldPdf[1, :, -1]$
    $+oldPdf[3, :, -1]$

  pdf[4, :, -1] $\leftarrow$ $oldPdf[2, :, -1]$ ▷ Bounce back from top
  pdf[7, :, -1] $\leftarrow$ $oldPdf[5, :, -1] - \frac{1}{6} \times wall\_velocity \times \rho\_wall$
  pdf[8, :, -1] $\leftarrow$ $oldPdf[6, :, -1] + \frac{1}{6} \times wall\_velocity \times \rho\_wall$
end procedure

Algorithm 6 Compute Density

procedure COMPUTEDENSITY(pdf)
  density $\leftarrow$ np.einsum("$ijk \rightarrow jk$", pdf) ▷ Compute density using probability density
  return density
end procedure

Algorithm 7 Compute Velocity

procedure COMPUTEVELOCITY(c, pdf)
  $density \leftarrow ComputeDensity()$ ▷ Compute the density using the probability density
  $velocity \leftarrow$ np.einsum("$ai, ixy \rightarrow xy$", c, pdf)$/density[x, y]$ ▷ Compute the velocity using the velocity set and probability density
  return *velocity*
end procedure

5. After updating the probability density function and applying boundary conditions (if any), the algorithm computes the density and velocity of the fluid based on the updated pdf. [Algorithm 6 and Algorithm 7]

6. Finally, if the collision flag is set to true, the algorithm performs the collision step. This step relaxes the probability density function towards equilibrium, effectively simulating the interactions and collisions between particles. [Algorithm 8]

Algorithm 8 LBM Collision Step

procedure CollisionStep
  equilibrium $\leftarrow$ E*quilibrium_Distribution*() ▷ Compute the equilibrium distribution function based on density and velocity
  pdf[i] $\leftarrow$ pdf[i] + omega $\times$ (equilibrium[i] $-$ pdf[i]) ▷ Relax towards equilibrium
end procedure

The main routine repeats these steps for the specified number of total steps, advancing the simulation of the fluid flow over time.

Different operations used:

`np.roll()`[3]: The operation `np.roll(array, shift, axis)` is used in the `streaming()` step to shift the pdf values for each velocity direction according to the velocity set c. The `axis` argument specifies the axis along which the elements are shifted, and the `shift` argument specifies the number of positions by which elements are to be shifted. In the context of the LBM simulation, this function is used to implement the streaming step, where the pdf values are shifted according to the predefined velocity set c.

For example, for the i-th velocity direction, the pdf values are shifted by $c[0, i]$ along the x-axis and by $c[1, i]$ along the y-axis, effectively moving the particles in the direction of the respective velocity components. This shifting operation is essential for particle propagation between neighboring grid points during the streaming step, allowing the LBM simulation to capture fluid flow behavior accurately.

`np.einsum()`[4]: The operation `np.einsum()` (Einstein summation) is a powerful tool for performing various array operations, including summation, contraction, and broadcasting. It allows us to specify how arrays should be combined using subscript-like notations. In the provided LBM code, `np.einsum()` is used for different purposes, as follows:

- `np.einsum("ijk->jk", pdf)`: This expression computes the summation of the pdf (probability density function) values over the first axis (i-axis). It effectively calculates the density at each grid point by summing the pdf values over all the velocity directions.
- `np.einsum("ai,ixy->axy", c, pdf)`: This expression performs a contraction operation between the velocity set c and the pdf *pdf*. It calculates the velocity components (u and v) at each grid point by combining the pdf values with the velocity set.
- `np.einsum("i,xy->ixy", weights, density)`: This expression performs an outer product-like operation between the weights `weights` and the density `density` (calculated in step 1). It calculates the equilibrium distribution for each velocity direction at each grid point.

In each case, `np.einsum()` efficiently handles the array operations, making the LBM simulation code concise and more readable. The ability to specify custom contraction patterns using subscript-like notations provides flexibility and ease of implementation for complex array manipulations often required in scientific computing tasks.

[3] https://numpy.org/doc/stable/reference/generated/numpy.roll.html
[4] https://numpy.org/doc/stable/reference/generated/numpy.einsum.html

3.2 Parallel Routine

To efficiently handle large-scale simulations, 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. The collision step is spatially local and does not require communication between processes [12], resulting in significant performance improvements and making it ideal for parallelization.

In this parallel implementation using MPI, each process is assigned a unique identifier known as **"rank"** within the communicator. The communicator (denoted as **"comm"** in the pseudocode) represents a group of processes capable of communicating with each other. These ranks play an important role in dividing the computational domain and facilitating communication between neighbouring processes.

The number of processes in each direction is user-defined. These values represent the number of ranks in the x and y directions. The user-defined values should adhere to the condition **if "num_procs_x" * "num_procs_y" == "size" or the total number of processes**, otherwise a **ValueError** will be raised.

Note: In the code repository[5], for the parallel simulation of Sliding Lid, the number of processes is user defined. But in case of the scaling test a factorization is automatically done from the total number of processes provided. This is done in order to keep the implementation generic and fast.

Subsequently, the domain is divided into smaller subdomains based on the user-defined number of processes in x and y direction. Each subdomain is assigned to a unique rank, and independent computation is performed within these subdomains. While the collision step inherently supports parallelism, communication becomes crucial during the streaming step, where particles move between neighbouring subdomains. To address this, additional layers of cells called **"ghost regions"** are introduced around each subdomain. These ghost cells store relevant data from adjacent processes, enabling accurate data exchange across domain boundaries. The implementation involves four communication steps for each process to exchange data at the domain boundaries. Data is sent and received using the **"Sendrecv"** function from the mpi4py library. The communication routine, (denoted as **"perform_communication"** in Algorithm 9), handles data exchange between neighbouring cells, ensuring the streaming step is executed accurately. The domain decomposition and communication strategy involve dividing the full two-dimensional lattice into spatial domains, each incorporating an additional ghost region. During communication, the outermost active lattice values are exchanged with the corresponding ghost lattice points in neighbouring domains. Figure 3.1 shows the communication process.

[5] https://github.com/ArkDy1312/High-Performance-Computing-with-Python---Fluid-Mechanics

Algorithm 9 Perform communication between neighboring cells to exchange boundary data.

```
function PerformCommunication
    Input:
        grid: The grid containing distribution functions.
        info: Information about the MPI process.
        comm: MPI communicator.
    Output:
        The updated grid after exchanging boundary data.
    if not info.boundaries_info.apply_right then
                                          ▷ Communication for the right boundary
        Create a receive buffer for the right boundary
        recv_buf ← grid[:, -1, :].copy()        ▷ Send the rightmost column to the neighboring
process on the right
        comm.Sendrecv(grid[:, -2, :].copy(), info.neighbors.right,
            recvbuf=recv_buf, sendtag=20, recvtag=21)
                                          ▷ Update the rightmost column with the received values
        grid[:, -1, :] ← recv_buf
    end if

    Similar blocks for the left, bottom, and top boundaries

    return grid
end function
```

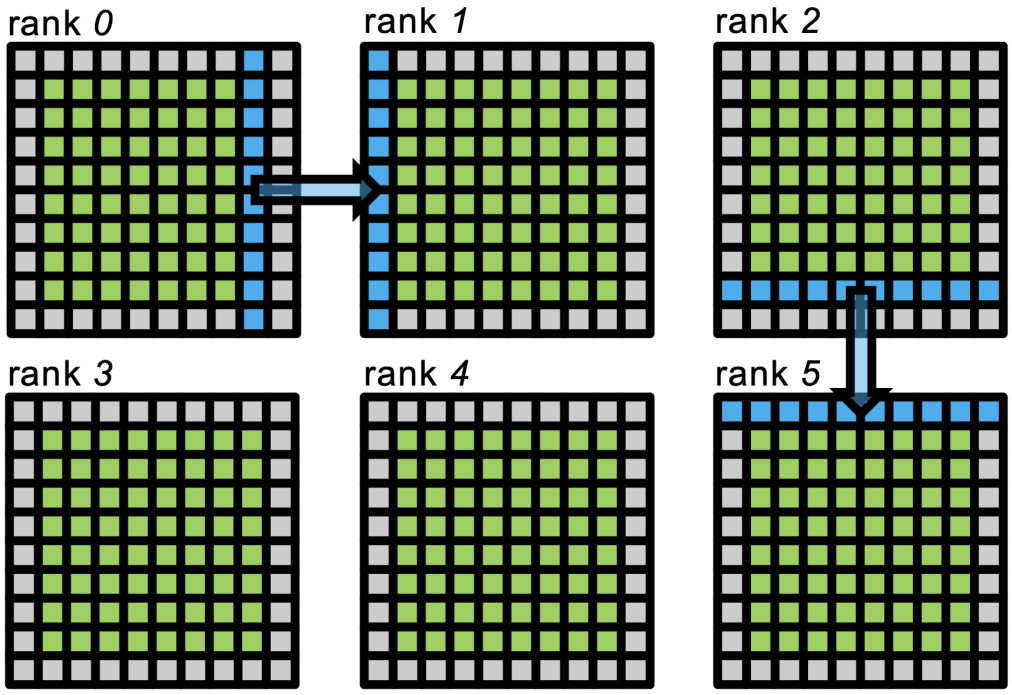

Figure 3.1: MPI domain decomposition and communication strategy. We start by dividing the domain into smaller subdomains with unique ranks (green lattice points). These green lattice points form the active physical domain for each rank. We introduce extra ghost cells, represented by grey lattice points, to serve as buffers. During communication, outermost green lattice points send data to adjacent outermost ghost points (blue arrows).[12].

Chapter 4

Numerical Results

Now we will validate the implementations, and also show the numerical results and visualisations of the experiments.

4.1 Shear Wave Decay

The shear wave decay implementation showcases the temporal evolution of a velocity perturbation within a flow. As viscosity diminishes flow velocity, it eventually converges to zero. This behaviour is initiated by introducing a sinusoidal velocity perturbation using the equation:

$$u_x(y, t = 0) = \begin{bmatrix} u_x(y, t = 0) \\ 0 \end{bmatrix} = \begin{bmatrix} \epsilon \sin\left(\frac{2\pi y}{Y}\right) \\ 0 \end{bmatrix} \tag{4.1.1}$$

The analytical solution for the temporal evolution of velocity is given by [13]:

$$u_x(y, t) = \epsilon \exp\left(-\nu(\frac{2\pi}{Y})^2 t\right) \sin\frac{2\pi y}{Y} \tag{4.1.2}$$

This analysis, rooted in Navier-Stokes equations, assumes insignificance of pressure and convection effects compared to viscosity. The validation process involves comparing the simulated and analytical results, illustrated in Figure 4.4 and Figure 4.5 for velocity. The agreement between these confirms the precision of rigid wall and moment updates. Figure 4.1 illustrates the density distribution over time, exhibiting convergence. Similarly, the simulation employs sinusoidal density in the x-direction, expressed as:

$$\rho(x, 0) = \rho_0 + \epsilon \sin\frac{2\pi x}{X} \tag{4.1.3}$$

where ρ_0 is a positive constant. We observe that this simulation also exhibits convergence.

Furthermore, the exponential decay of moment fluctuations is defined by [14; 15]:

$$Q_t(t) = \exp\left(-\nu(\frac{2\pi}{Y})^2 t\right) \tag{4.1.4}$$

where one of the moment quantities: $Q(x, t) = \epsilon Q_x(x) Q_t(t)$. Validation experiments substantiate the implementation by estimating viscosity through Equation 4.1.4. The analytical viscosity is computed using the formula [10] $\nu = c_s^2\left(\frac{1}{\omega} - \frac{1}{2}\right)$. Simulated viscosity is computed based on the decay curve of density and velocity. It should be noted that selecting ω values close to 0.0 or 2.0 leads to numerical instability. Figure 4.3 visually presents the experimental outcomes when choosing different omega values.

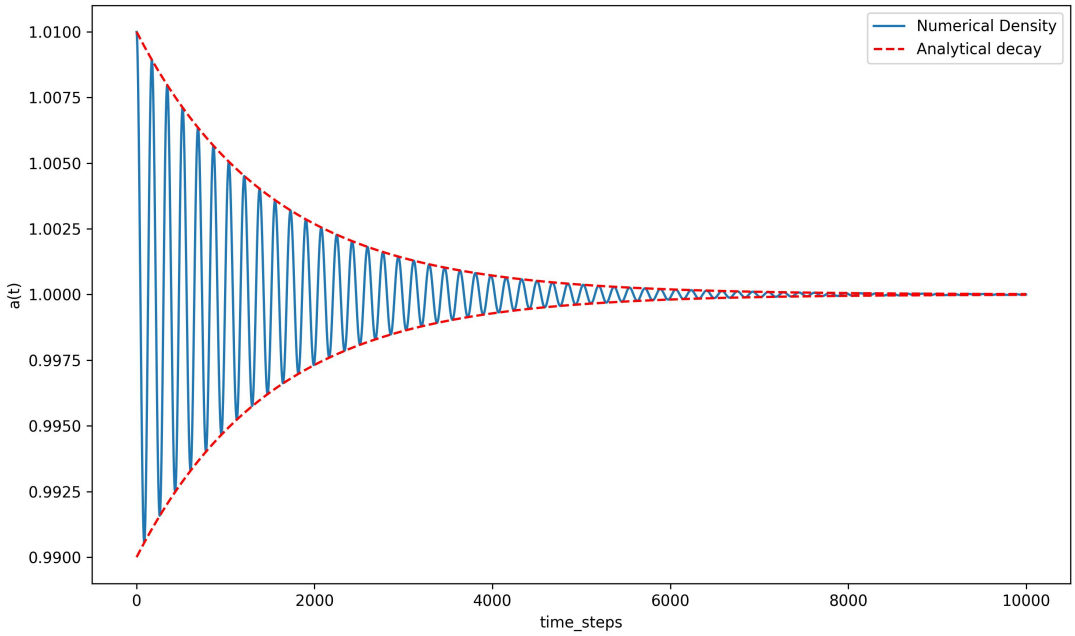

Figure 4.1: The time evolution of the sinusoidal density at $y = 25$ with the lattice grid size of $(100, 100)$. The coefficients ϵ and ρ_0 in Equation 4.1.3 are set to 0.01 and 1.0 and the velocity is initialised by $u(x, 0) = (0, 0)$. The relaxation term ω is set to 1.0.

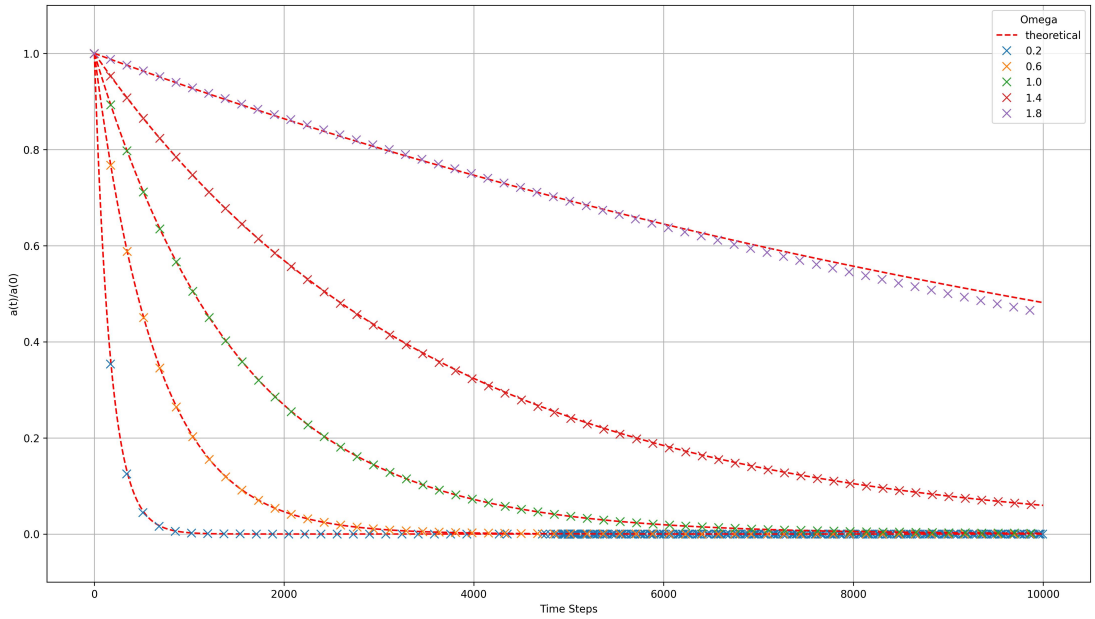

Figure 4.2: Time evolution of the amplitude of perturbation for different ω vs their theoretical values.

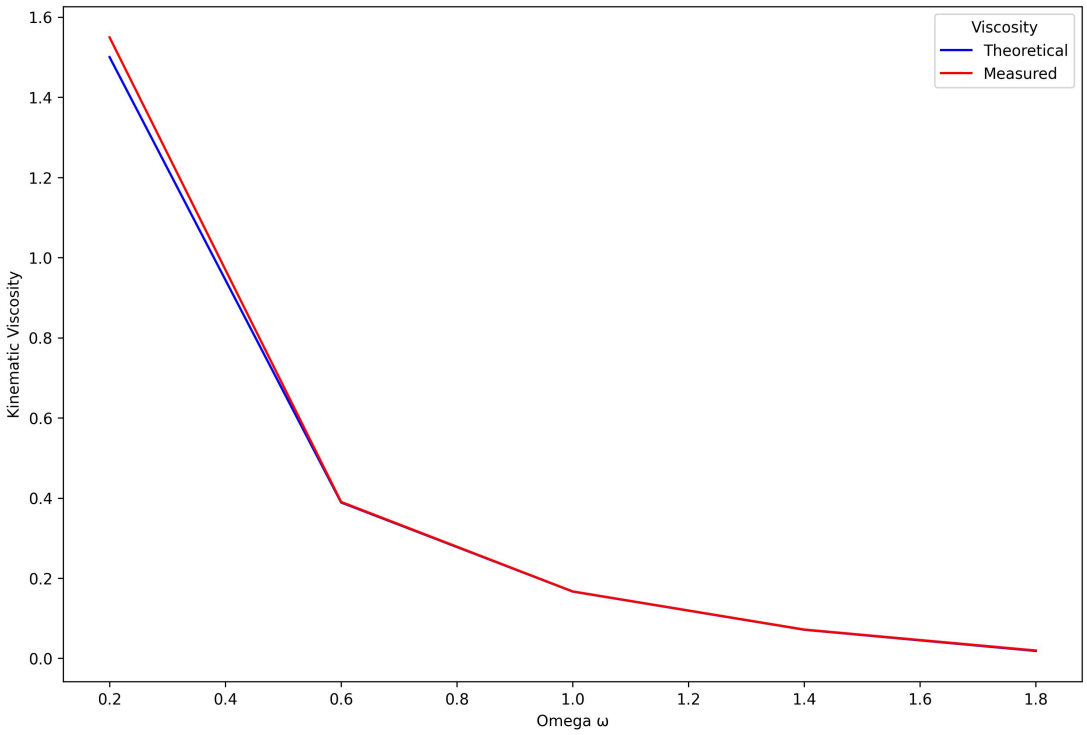

Figure 4.3: Theoretical Kinematic Viscosity vs Simulated Kinematic Viscosity for different ω.

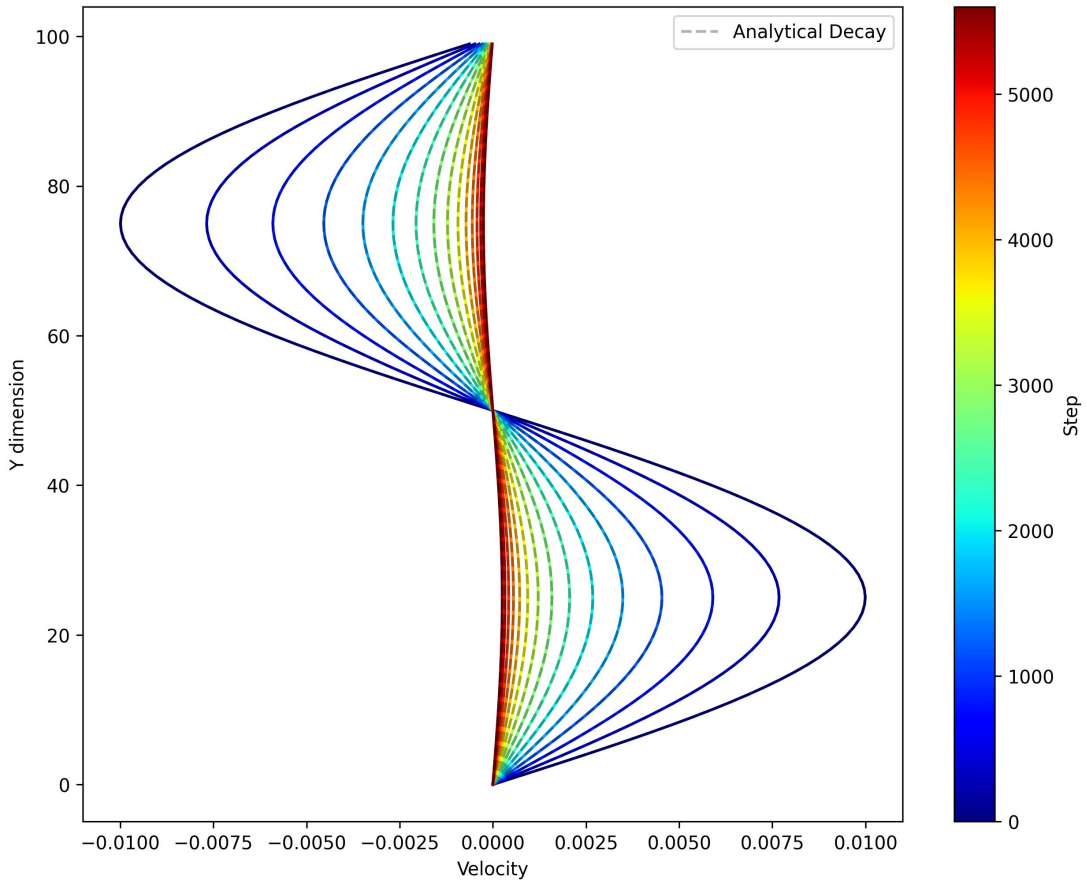

Figure 4.4: The time evolution of the simulated sinusoidal velocity at $x = 25$ with the lattice grid size of $(100, 100)$. The coefficient ϵ is set to 0.01 and the initial density is set to 1.0. The relaxation term ω is set to 1.0. The plot also shows the analytical decay.

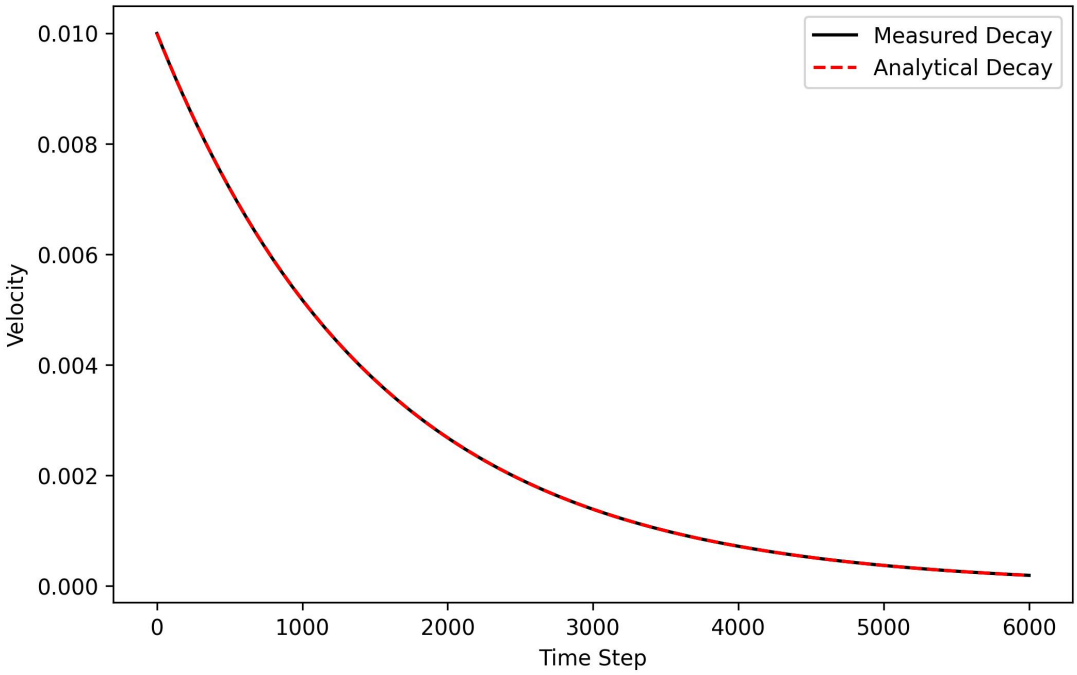

Figure 4.5: The time evolution of the simulated sinusoidal velocity at $x = 25$ and $y = 50$ with the lattice grid size of $(100, 100)$. The coefficient ϵ is set to 0.01 and the initial density is set to 1.0. The relaxation term ω is set to 1.0. The plot also shows the analytical decay.

4.2 Couette Flow

The Couette flow involves a fluid flow confined between two walls: one remains stationary while the other moves horizontally with a velocity of U_w. This motion arises due to the exertion of a viscous drag force on the fluid. For this simulation we utilise the bounce-back boundary conditions at the moving and stationary walls, and periodic boundary conditions at the inlet and outlet.

The analytical solution for this configuration permits the validation of the dynamic wall implementation.The analytical expression is formulated as [16]:

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

where Y represents the distance between the two walls, and $u_x(\cdot, y)$ signifies the horizontal fluid velocity at a specific position (x, y). Figure 4.7 visually demonstrates that the velocity flow has iteratively converged towards the analytical solution. The alignment becomes near-perfect over time when the velocity stabilises, thus successfully validating the moving wall implementation. It should be noted that $u_x(\cdot, y)$ is independent of x.

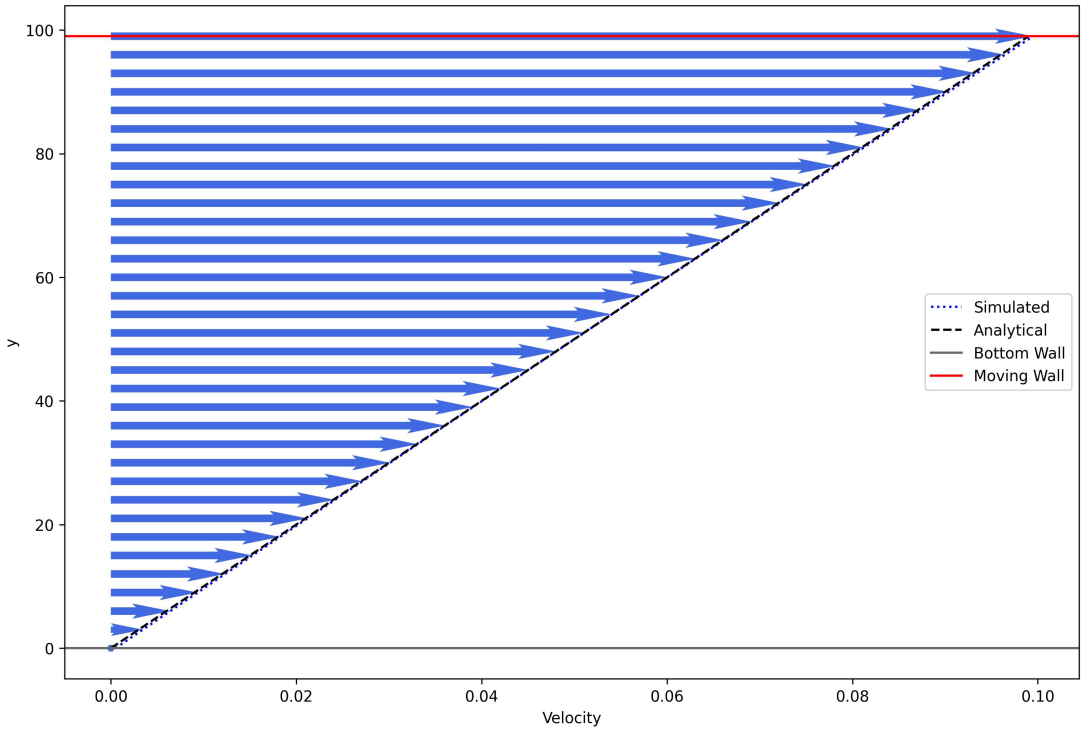

Figure 4.6: The velocity vectors at $x = 50$, for lattice grid size of $(100, 100)$ at time step $= 30000$. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. The relaxation term $\omega = 1.0$ and wall velocity $= 0.1$.

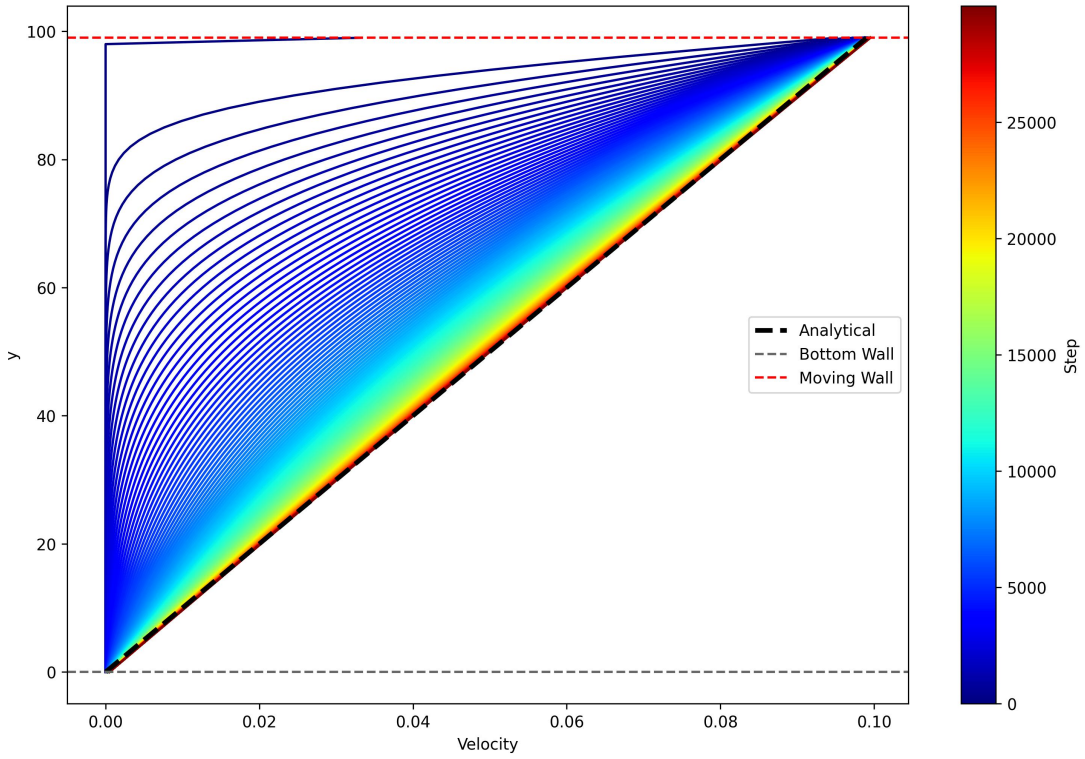

Figure 4.7: The velocity evolution at $x = 50$, for lattice grid size of $(100, 100)$ till time step = 30000. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. The relaxation term $\omega = 1.0$ and wall velocity = 0.1.

4.3 Poiseuille Flow

The Poiseuille flow entails a fluid flow confined between two stationary walls. This flow arises from a constant pressure difference $\frac{dp}{dx}$ applied in the horizontal direction between the two walls. During the experiment, we employ bounce-back boundary conditions at both the stationary and rigid walls, alongside the pressure PBC at the inlet and outlet.

This flow configuration also possesses an analytical solution, helping us to validate the pressure periodic boundary condition (PBC) implementation. The analytical expression is given by [17]:

$$u_x(\cdot, y) = -\frac{1}{2\nu\rho_{\text{avg}}}\frac{dp}{dx}y(Y - y)$$

where ρ_{avg} represents the average density of the fluid and ν denotes viscosity. Figure 4.10 illustrates the outcomes, showcasing the simulated results. As we can see, convergence (progressively) is also achieved here.

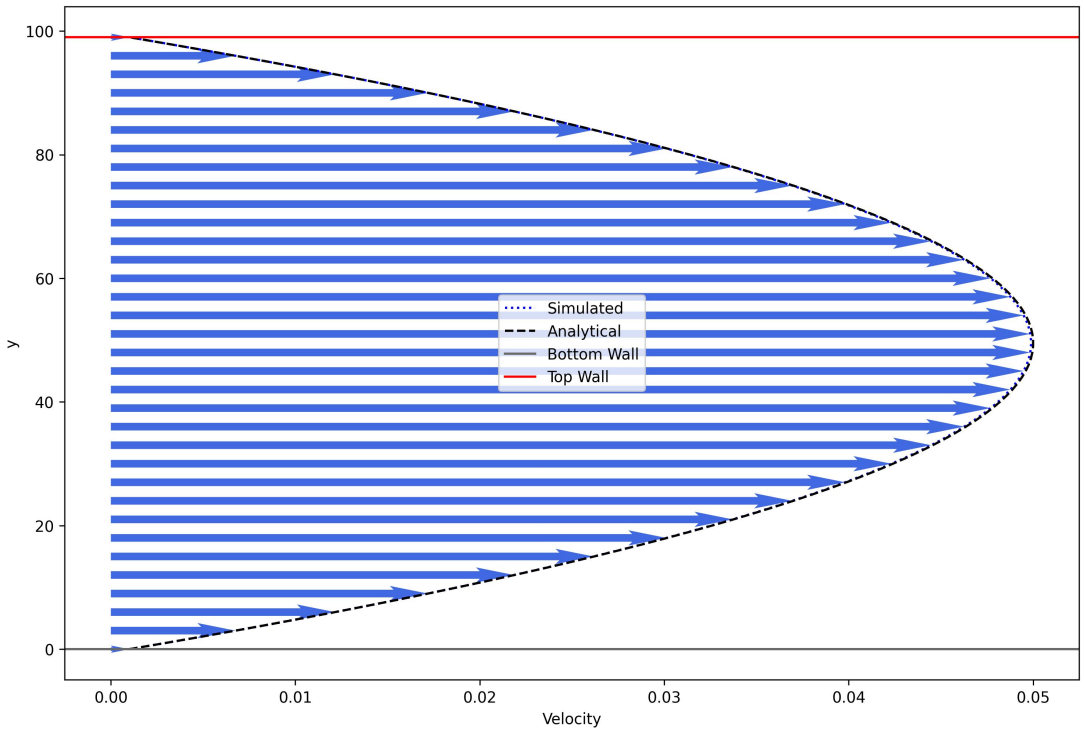

Figure 4.8: The velocity vectors at $x = 50$, for lattice grid size of $(100, 100)$ at time step = 45000. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. The relaxation term $\omega = 1.0$. ϵ is chosen to be 0.001, hence the inlet and outlet density are $\rho_{in} = 1.001$ and $\rho_{out} = 0.999$ respectively.

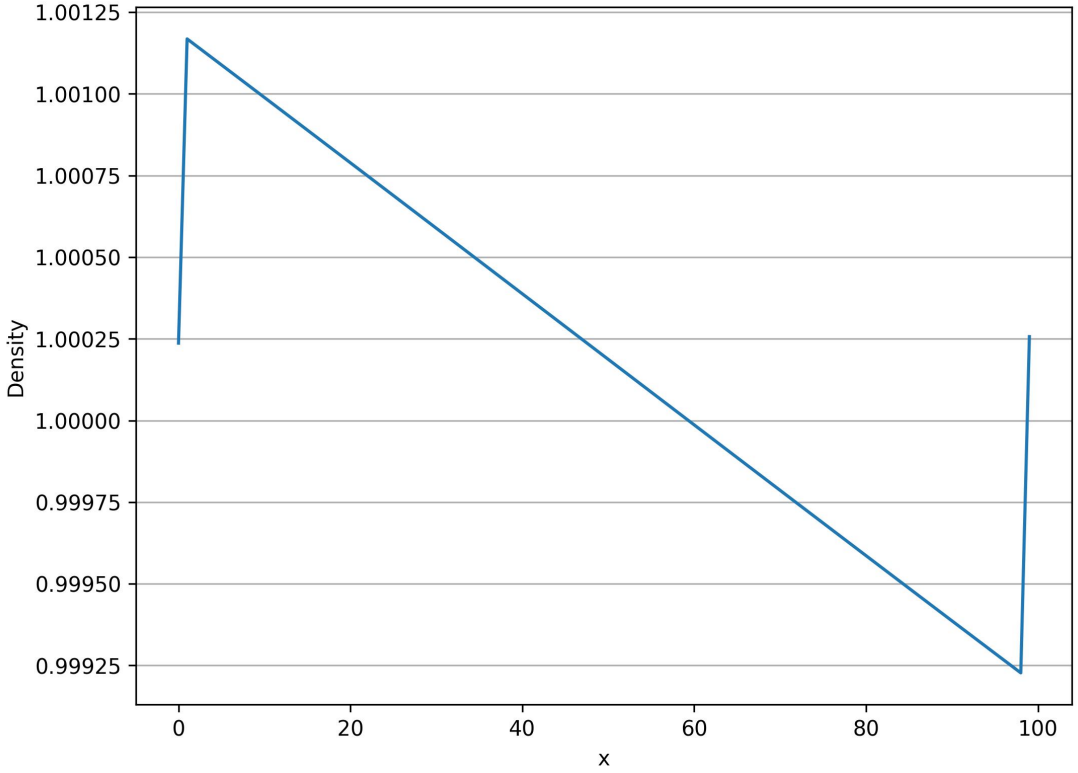

Figure 4.9: The density along the centerline of the channel $y = 50$, for lattice grid size of $(100, 100)$ at time step $= 45000$. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. The relaxation term $\omega = 1.0$. ϵ is chosen to be 0.001, hence the inlet and outlet density are $\rho_{in} = 1.001$ and $\rho_{out} = 0.999$ respectively.

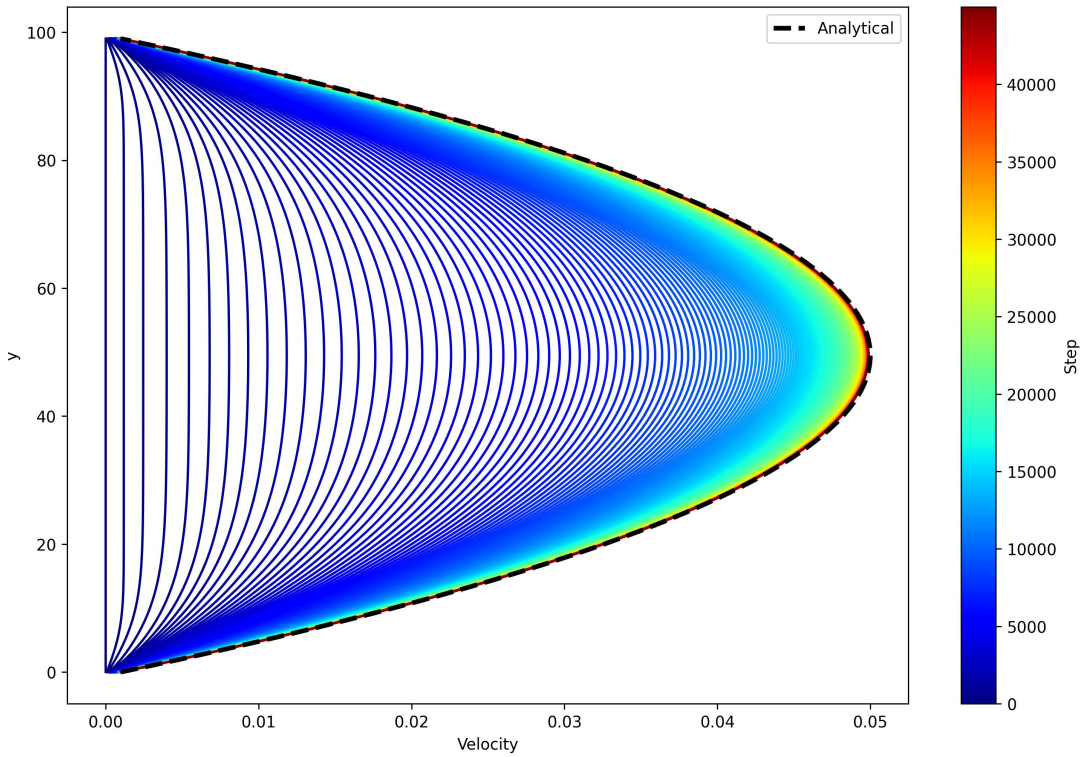

Figure 4.10: The velocity evolution at $x = 50$, for lattice grid size of $(100, 100)$ till time step $=$ 45000. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. The relaxation term $\omega = 1.0$. ϵ is chosen to be 0.001, hence the inlet and outlet density are $\rho_{in} = 1.001$ and $\rho_{out} = 0.999$ respectively.

4.4 Lid-Driven Cavity

In this section, we delve into a practical example, simulating the lid-driven cavity. We model the flow within a box featuring three stationary walls and one moving wall, which acts as the lid. The Reynolds number is denoted by:

$$Re = \frac{LU}{\nu}$$

where L represents the characteristic length parameter and U denotes the moving wall velocity. Notably, turbulence arises when the Reynolds number surpasses 1000 [18]. Two flow systems become dynamically similar if their Reynolds numbers and geometries are analogous [19]. To illustrate this, we provide results with different viscosities (ν) and wall velocity (U_w) configurations that adhere to the Reynolds number of 1000, under the conditions $L = X = Y = 300$, as shown in Figure 4.13. The figures showcase the convergence to similar flow patterns, aligning with the key property of the Reynolds number.

Additionally, Figure 4.12 portrays the temporal evolution of the streaming plot with a Reynolds number of 1000. The different plots illustrate gradual changes in the streaming behaviour, accompanied by the emergence of spirals at corners due to turbulence.

	Serial Run	Parallel Run	Absolute Differences
Maximum	0.09766	0.09766	0.00000
Minimum	-0.06697	-0.06697	0.00000
Absolute Sum	2687.53874	2687.53874	0.00000

Table 4.1: Velocity results comparison between Serial and Parallel Runs. The different parameter values chosen for this test are Lattice Size = (300, 300), Number of time steps = 100000 and Reynolds number = 1000.

For the parallel version of this experiment, we employ MPI with 8 processes, as demonstrated in Table 4.1 for parallel implementation validation using the above conditions. The absolute error summation of velocity across the entire domain is found to be 0.0, affirming the parallel implementation is equivalent to the serial counterpart. The test code for checking whether the serial and parallel implementations are similar can be found in the repository at `tests/test_parallel_vs_serial.py`.

It's worth noting that the serial version of this experiment takes almost 3x the time as compared to the parallel one. For instance, running a simulation serially on an Apple M1 8-core CPU with 8GB RAM takes about 18 minutes to complete, while the parallel version takes roughly 7 minutes to complete. The advantage of the Lattice Boltzmann Method lies in its parallel computational capabilities. Therefore, we assess the scalability of this simulation by employing various numbers of processes. These scalability experiments were conducted on the bwUniCluster 2.0[1]. Each thread is assigned to a single processor, aligning with the approach outlined in Section 3.2.

The relationship between Million Lattice Updates Per Second (MLUPS) and the number of processes is shown in Figure 4.14. Evidently, larger grid sizes correspond to lower MLUPS values with fewer processes due to higher load concentration. However, when the number of processes increases, simulations featuring larger domains exhibit increased efficiency. It's worth noting that while ideal linear growth in MLUPS with respect to the number of processes is anticipated, the observed slowdowns from approximately

$$\frac{X \times Y}{100}$$

processes (as depicted in Figure 4.14) result from communication latency and synchronisation waits, in line with Amdahl's law [20].

Amdahl's law provides a theoretical framework to assess the potential speedup when enhancing a portion of a computation. Formulated by Gene Amdahl in 1967, the law posits that the

[1] https://wiki.bwhpc.de/e/BwUniCluster2.0

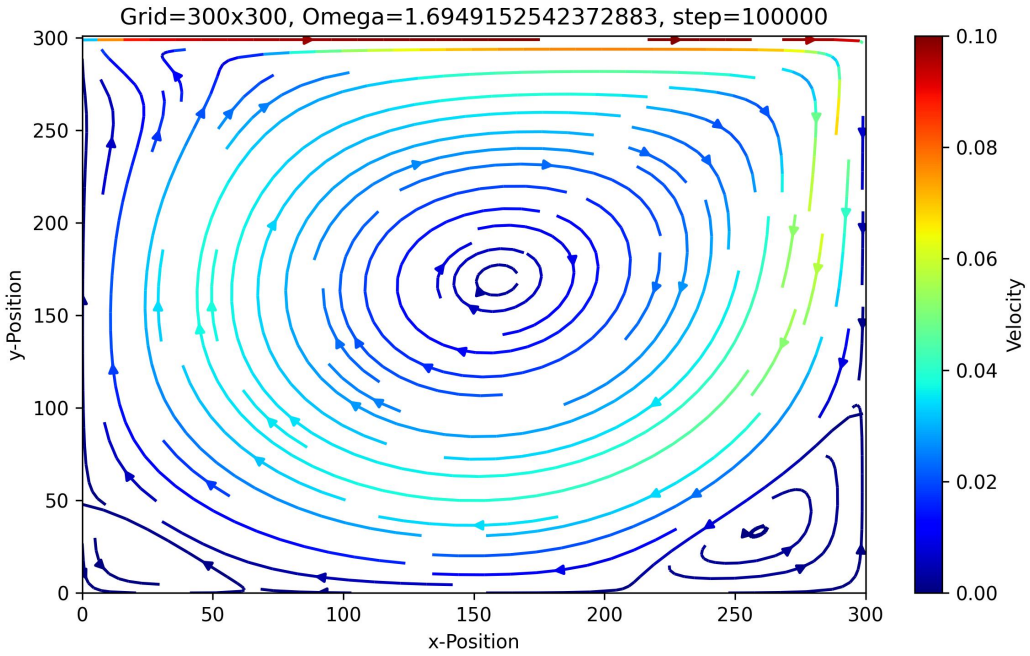

Figure 4.11: The stream plot of the Sliding Lid simulation at T = 100000 with the lattice grid size of $(300, 300)$, the Reynolds number is chosen to be 1000 and the wall velocity of 0.1 is chosen. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$.

speedup achievable through parallelization is limited by the sequential component of the computation. Specifically, if f represents the fraction of the computation that cannot be parallelized, the potential speedup S is given by: Theoretical speedup of the execution of the whole task [21]:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

s is the speedup of the part of the task that benefits from improved system resources and p is the proportion of execution time that the part benefiting from improved resources originally occupied. As the number of processes increases, the impact of the non-parallelizable portion diminishes, but it can never be completely eliminated.

Amdahl's law underscores the significance of identifying and optimising critical sequential sections in parallel algorithms, as the overall speedup is influenced by the inherent sequential bottlenecks. This principle serves as a valuable guideline when designing and scaling parallel simulations, as observed in the context of our lid-driven cavity experiment.

Note: The code repository[2] contains instructions on how to run the serial, parallel versions of the Sliding Lid/Lid-driven Cavity simulation and also how to run the scaling tests.

[2] https://github.com/ArkDy1312/High-Performance-Computing-with-Python---Fluid-Mechanics

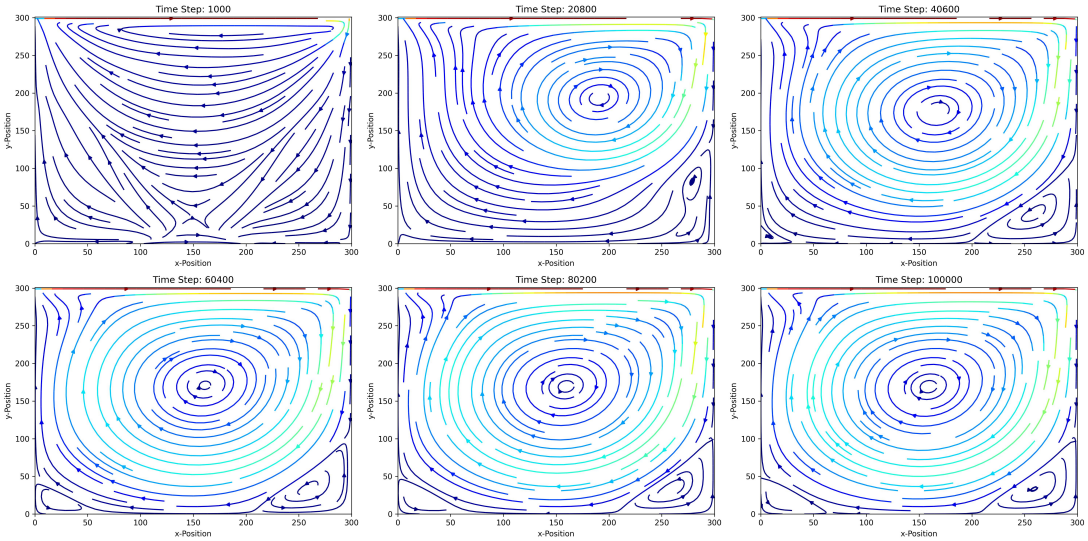

Figure 4.12: The time evolution of the stream plots of the Sliding Lid simulation with the lattice grid size of $(300, 300)$, the Reynolds number is chosen to be 1000 and the wall velocity of 0.1 is chosen. We run the simulation for T = 100000 times. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$.

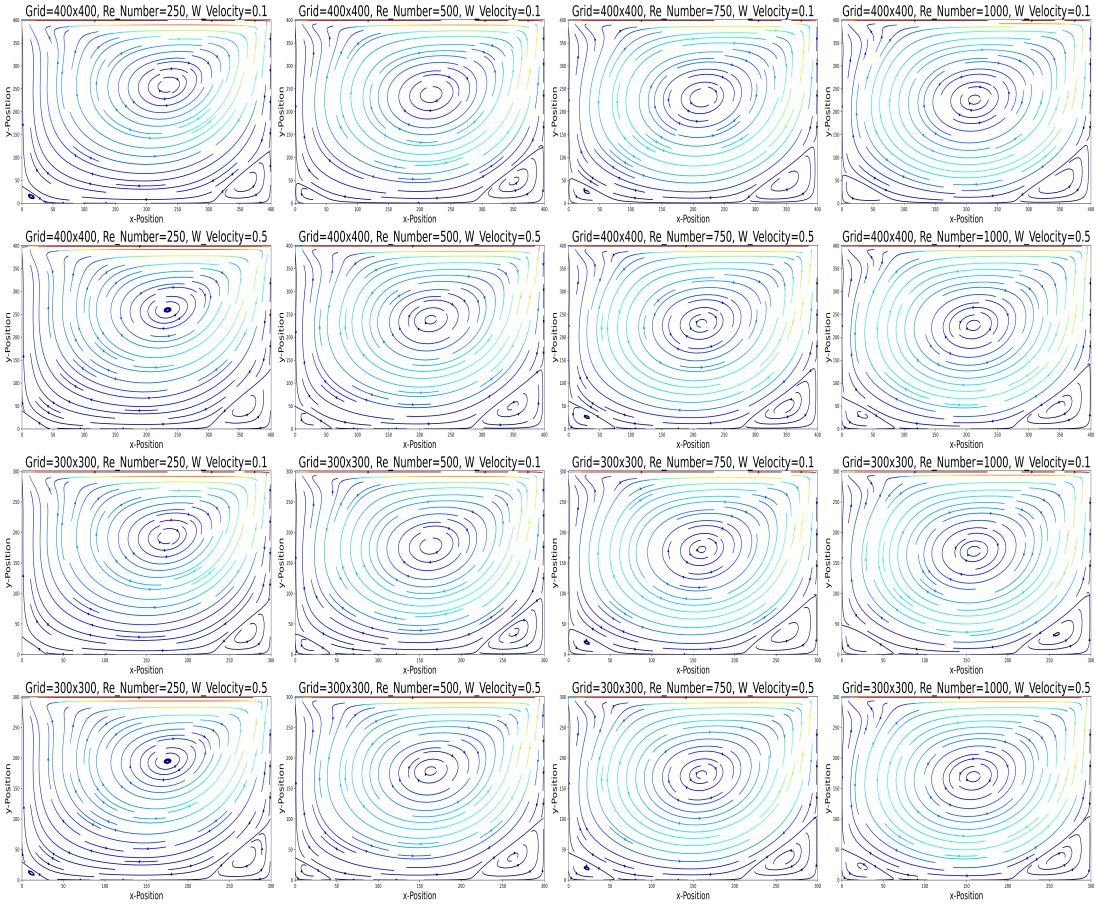

Figure 4.13: The stream plots of the Sliding Lid simulation with the lattice grid sizes of $(300, 300)$, $(400, 400)$, the Reynolds number are chosen to be 250, 500, 750, 1000 and the wall velocity are chosen to be 0.1, 0.5. We run the simulation for T = 100000 times for each setting. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$.

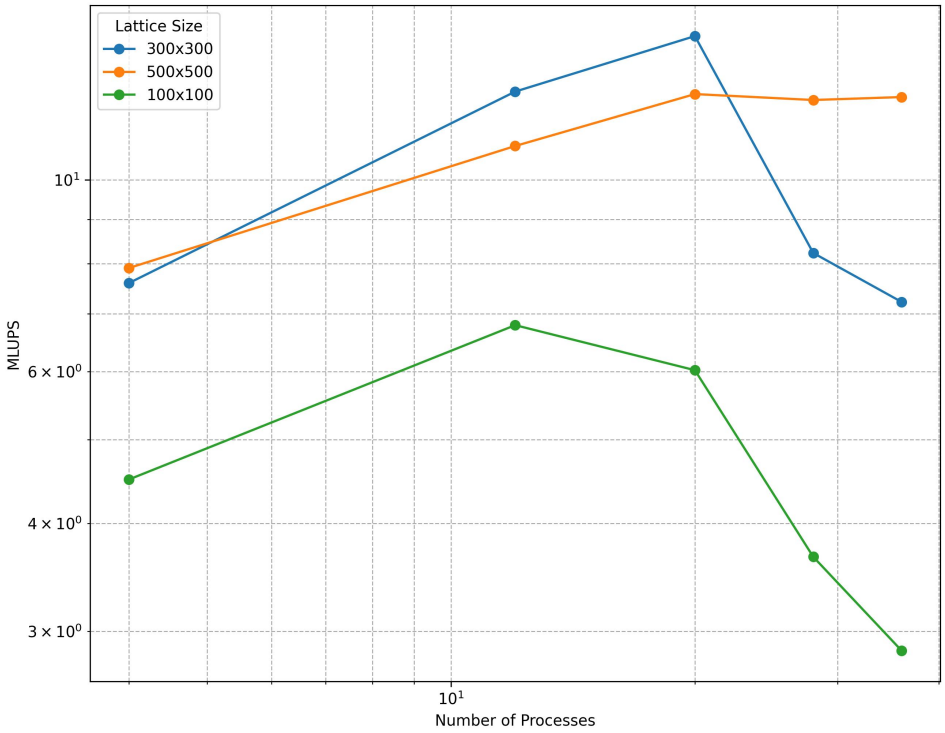

Figure 4.14: The scaling test of the Sliding Lid simulation. The lattice grid sizes of $(100, 100)$,$(300, 300)$ and $(500, 500)$. The numbers of processes are $(4, 12, 20, 28, 36)$. The viscosity and the wall velocity are set to $\nu = 0.03$ and 0.1 and we capture the MLUPS value at T = 10000 step. The initial density and velocity are $\rho(x) = 1.0$, $u(x) = (0, 0)$. Both axes are log-scale.

Chapter 5

Conclusion

This paper delves into both the theoretical underpinnings of the Lattice Boltzmann Method (LBM) and its practical implementations.

We started out by highlighting the transformative role of High-Performance Computing (HPC) in advancing scientific and engineering domains. The Lattice Boltzmann Method (LBM), a powerful technique for simulating fluid dynamics, was introduced with a focus on its parallelization using the Message Passing Interface (MPI). LBM's unique lattice-based grid structure, versatility in handling complex geometries, and intrinsic ability to model mesoscopic phenomena were discussed. The broad applications of LBM, including porous media flows, microfluidics, and multiphase interactions, were underscored. Leveraging HPC resources through MPI parallelization not only accelerates LBM simulations but also enables finer resolutions for more accurate results.

Then we delved into the Lattice Boltzmann Method (LBM), a robust technique for simulating fluid flows. It covers fundamental equations, discretization, moment updates, and boundary handling. LBM employs BTE to track particle evolution, while discretization and collision steps ensure computational feasibility. Moment updates translate to macroscopic properties, and boundary handling methods like bounce-back and PBC simulate interactions with walls.

In Chapter 3, we delienated the systematic implementation of the Lattice Boltzmann Method (LBM) and its parallelization techniques for computational efficiency. The core focus is on translating LBM equations into Python code to simulate fluid dynamics accurately. The implementation assumes D2Q9 lattice discretization and employs Numpy and mpi4py libraries. The serial routine encompasses steps such as computing equilibrium distribution functions, handling boundary conditions, streaming particles, updating density and velocity, and executing the collision step. These steps repeat for the simulation duration. Parallelization involves spatial domain decomposition with MPI. The domain is divided into subdomains, assigned to different MPI processes. Communication is essential during streaming; ghost regions facilitate data exchange across domain boundaries. This strategy enables efficient large-scale fluid dynamics simulations.

Finally, Chapter 4 validates our implementations and presents numerical results for diverse experiments. **Shear Wave Decay:** The initial experiment showcases the convergence of velocity and density perturbations within a flow. We simulate sinusoidal perturbations, compare results with analytical solutions, and explore moment fluctuations. **Couette Flow:** Next, we simulate Couette flow and validate our dynamic wall treatment by comparing simulation results with analytical solutions. **Poiseuille Flow:** We examine Poiseuille flow, validate pressure periodic boundary conditions, and confirm accuracy by comparing simulated profiles with analytical solutions. **Lid-Driven Cavity:** In a practical example, we simulate a lid-driven cavity, validating the simulation against theoretical expectations and showcasing parallel implementation efficiency. We discuss scalability and the role of Amdahl's law.

Chapter 6

References

[1] J. Boon, "The lattice boltzmann equation for fluid dynamics and beyond," *European Journal of Mechanics - B/Fluids*, vol. 22, p. 101, 01 2003.

[2] M. D. Chen, S. and R. Mei, "Lattice boltzmann model for simulation of liquid-vapor-solid three-phase flows," *Journal of Statistical Physics*, vol. 92, no. 3-4, pp. 477–505, 1998.

[3] C. K. Aidun and J. R. Clausen, "Lattice-boltzmann method for complex flows," *Annual Review of Fluid Mechanics*, vol. 42, no. 1, pp. 439–472, 2010.

[4] X. Shan and H. Chen, "Lattice boltzmann model for simulating flows with multiple phases and components," *Phys. Rev. E*, vol. 47, pp. 1815–1819, Mar 1993.

[5] Z. Guo and C. Shu, *Lattice Boltzmann Method and Its Applications in Engineering*. Advances in computational fluid dynamics, World Scientific, 2013.

[6] T. Sterling, M. Brodowicz, and M. Anderson, *High Performance Computing: Modern Systems and Practices*. Elsevier Science, 2017.

[7] P. L. Rossiter, *The Electrical Resistivity of Metals and Alloys*. Cambridge Solid State Science Series, Cambridge University Press, 1987.

[8] "Non-equilibrium thermodynamics of the solidification problem," *Journal of Crystal Growth*, vol. 66, no. 3, pp. 575–585, 1984.

[9] G. Zhao-Li, Z. Chu-Guang, and S. Bao-Chang, "Non-equilibrium extrapolation method for velocity and pressure boundary conditions in the lattice boltzmann method," *Chinese Physics*, vol. 11, p. 366, apr 2002.

[10] T. Krueger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. Viggen, *The Lattice Boltzmann Method: Principles and Practice*. Graduate Texts in Physics, Springer, 2016.

[11] S. Succi, *The Lattice Boltzmann Equation: For Complex States of Flowing Matter*. 06 2018.

[12] L. Pastewka and A. Greiner, *HPC with Python: An MPI-parallel Implementation of the Lattice Boltzmann Method*. Universitätsbibliothek Tübingen, 2019.

[13] L. Fei, K. Luo, and Q. Li, "Three-dimensional cascaded lattice boltzmann method: Improved implementation and consistent forcing scheme," *Physical Review E*, vol. 053309, pp. 1–12, 05 2018.

[14] B. Hess, "Determining the shear viscosity of model liquids from molecular dynamics simulations," *The Journal of Chemical Physics*, vol. 116, pp. 209–217, 01 2002.

[15] B. J. Palmer, "Transverse-current autocorrelation-function calculations of the shear viscosity for molecular liquids," *Phys. Rev. E*, vol. 49, pp. 359–366, Jan 1994.

[16] P. Nagy-György and C. Hős, "A graphical technique for solving the couette-poiseuille problem for generalized newtonian fluids," *Periodica Polytechnica Chemical Engineering*, vol. 63, 05 2018.

[17] A. Mendiburu, L. Carrocci, and J. Carvalho, "Analytical solution for transient onedimensional couette flow considering constant and time-dependent pressure gradients," *Revista de Engenharia Térmica*, vol. 8, p. 92, 10 2018.

[18] T. P. Chiang, W.-H. Sheu, and R. R. Hwang, "Effect of reynolds number on the eddy structure in a lid-driven cavity," *International Journal for Numerical Methods in Fluids*, vol. 26, pp. 557–579, 1998.

[19] P. Kundu, I. Cohen, and D. Dowling, *Fluid Mechanics*. Science Direct e-books, Elsevier Science, 2012.

[20] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), p. 483–485, Association for Computing Machinery, 1967.

[21] Wikipedia contributors, "Amdahl's law — Wikipedia, the free encyclopedia," 2023.