High Performance Computing with Python

Final Report

Shuhei Watanabe

5171091

watanabs@informatik.uni-freiburg.de

August 5, 2021

Contents

1

Introduction

Large-scale physics experiments often require large budgets and it is hard to perform experiments with several different parameters. For this reason, many research has been performed to simulate real-world phenomenon. One of the phenomenon is fluid flow. For example, fluid flow simulations allow us to deeply understand how the car body shape relates to the aerodynamic drag and to optimize the car design through the simulations with various designs rather than making real cars [1].

Such simulations require a scheme to simulate the physical states at each time step and the lattice Boltzmann method (LBM) [2] is one of the well-known schemes for the fluid flow simulation method. LBM approximates the physical states of a myriad of microscopic particles, i.e. usually obtained by solving the Navier-Stokes equation, by mesoscale physical states at each lattice grid. The physical states or **moments** are iteratively simulated based on the Maxwell velocity distribution function [3] and the fluid flow at each time step is derived from the moments.

The major advantages of LBM are the followings:

- **Simple implementation**: The governing equations of each moment are simple and the collision handling only considers the adjacent lattices.
- **Parallelization**: The parallelization scales well due to the local dynamics nature of LBM [4]

For those reasons, LBM is one of the most successful methods and we would like to introduce LBM in detail in this paper. The paper structure is as follows:

1. **Lattice Boltzmann method (LBM)** : Show the governing equations and provide pseudocodes to promote the understandings
2. **Numerical results** [1]: Provide how we can validate the implementations, and how effective the parallel computation is

All the codes follow `pep8 style` [2] and are tested using `unittest` [3]. Furthermore, **the step-by-step reproduction instruction is available** in `README.md` on this repository.

[1] The code is available at: https://github.com/nabenabe0928/high-performance-computing-fluid-dynamics-with-python

[2] https://www.python.org/dev/peps/pep-0008/

[3] https://docs.python.org/3/library/unittest.html

2

Lattice Boltzmann method

In this chapter, we describe how the equations used in LBM are derived. More specifically, we explain the **Boltzmann transport equation (BTE)** [5], i.e. the basic equations of the kinetic theory of gases and how to handle the boundary conditions.

2.1 The Boltzmann transport equation (BTE)

The BTE formulates the time evolution of the particle probability density $f(\boldsymbol{x}, \boldsymbol{u}, t)$ given the velocity $\boldsymbol{u}$ and the position $\boldsymbol{x}$ of particles. The BTE relaxes the particle distribution to the Maxwell velocity distribution function [3] and the approximation of the relaxation of f towards f^{eq} is described as follows [6]:

$$\frac{df(\boldsymbol{x}, \boldsymbol{u}, t)}{dt} = -\frac{f(\boldsymbol{x}, \boldsymbol{u}, t) - f^{\text{eq}}(\boldsymbol{u}; \rho(\boldsymbol{x}, t), \boldsymbol{u}(\boldsymbol{x}, t), T(\boldsymbol{x}, t))}{\tau} \tag{2.1}$$

where f^{eq} is statistical equilibrium, $T(\boldsymbol{x}, t)$ is the temperature at $\boldsymbol{x}$ of time step t and τ is a characteristic time. The characteristic time determines how quickly the fluid converges towards equilibrium. The higher τ yields the slower convergence towards the equilibrium. Eq (2.1) is used for the update of the particle probability density function. Furthermore, this particle probability density function $f(\boldsymbol{x}, \boldsymbol{u}, t)$ is used for computing the physical states of the fluid, such as density and velocity. The momentum updates are performed via [7]:

$$\rho(\boldsymbol{x}, t) = \int f(\boldsymbol{x}, \boldsymbol{u}, t) d\boldsymbol{u}, \ \boldsymbol{u}(\boldsymbol{x}, t) = \frac{1}{\rho(\boldsymbol{x}, t)} \int \boldsymbol{u} f(\boldsymbol{x}, \boldsymbol{u}, t) d\boldsymbol{u} \tag{2.2}$$

The underlying equations allow simulating fluid flow as seen in the latter parts of this paper.

2.2 Time-step update of the BTE

The aforementioned BTE is formulated in the continuous domain; therefore, we need to discretize spatially and temporally to make the computation feasible by simulations. In this paper, we focus on discretization in two-dimensional space. The discretization for space and time is performed so that the equality condition of the following inequality (Courant-Friedrichs-Lewy condition) holds [8, 9]:

$$\boldsymbol{c}_i \Delta t \leq ||\Delta \boldsymbol{x}_i|| \tag{2.3}$$

where Δt is the time step size and $\Delta \boldsymbol{x}_i$ is the distance between the closest grid in the direction of $\boldsymbol{c}_i$ that is defined by:

$$\boldsymbol{c} = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix}^\top \tag{2.4}$$

Note that this specific discretization in two-dimensional space with nine directions shown in Figure 2.1 is called D2Q9. In this setting, we first discretize the particle probability density function in the nine directions by subscripting as $f_i(\boldsymbol{x}, t)$. Then Eq (2.2) becomes the followings:

$$\rho(\boldsymbol{x}, t) = \sum_i f_i(\boldsymbol{x}, t), \;\; \boldsymbol{u}(\boldsymbol{x}, t) = \frac{1}{\rho(\boldsymbol{x}, t)} \sum_i \boldsymbol{c}_i f_i(\boldsymbol{x}) \tag{2.5}$$

Note that we regard the density as a unit molecular mass in Eq (2.5). Additionally, the equilibrium in Eq (2.1) is computed as:

$$\underbrace{f_i(\boldsymbol{x} + \boldsymbol{c}_i \Delta t, t + \Delta t) - f_i(\boldsymbol{x}, t)}_{\text{streaming}} = \underbrace{-\omega \Big[f_i(\boldsymbol{x}, t) - f_i^{\text{eq}}(\boldsymbol{x}, t) \Big]}_{\text{collision}} \tag{2.6}$$

where $\omega = \Delta t / \tau$ is the relaxation parameter. The equilibrium is computed as [10]:

$$f_i^{\text{eq}}(\boldsymbol{x}, t) = w_i \rho(\boldsymbol{x}, t) \left[1 + 3\boldsymbol{c}_i \cdot \boldsymbol{u}(\boldsymbol{x}, t) + \frac{9}{2} (\boldsymbol{c}_i \cdot \boldsymbol{u}(\boldsymbol{x}, t))^2 - \frac{3}{2} ||\boldsymbol{u}(\boldsymbol{x}, t)||^2 \right] \tag{2.7}$$

where the index i corresponds to Figure 2.1 and $\boldsymbol{w} = [\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}]$. In the streaming step, the grid receives the particle flow $f_i(\boldsymbol{x} + \boldsymbol{c}_i \Delta t, \cdot)$ from its nine adjacent grids. In the collision step, we relax the probability density function towards the equilibrium f_i^{eq} by considering the effects of the particle collision.

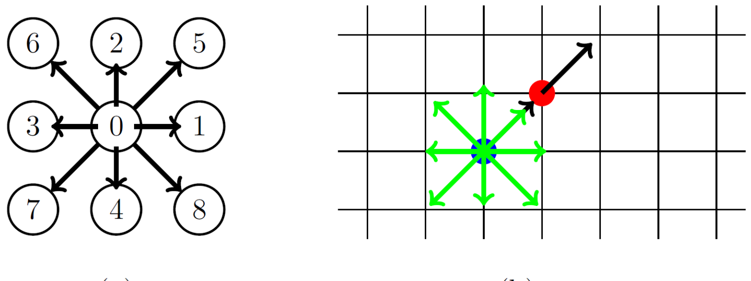


Figure 2.1: (a) The discretization on the velocity space according to D2Q9. (b) The uniform two-dimensional grids for the discretization in the physical space.

2.3 Boundary handling

In this section, we briefly discuss how we handle the particles that bump into boundaries. Note that the boundary handling is performed after the streaming step that is discussed in the previous section and we usually use the direction that is opposite to the direction i for the bounce-back.

For this reason, we will denote $f_i^\star$ as the i-th direction particle probability density function after the streaming step and $i^\star$ as the direction opposite, i.e. **reflected direction**, to i. Those directions follow D2Q9 illustrated in Figure 2.1. Additionally, there are the following two ways to implement the boundary conditions [11]:

1. **Dry nodes**: The boundaries are located on the link between nodes
2. **Wet nodes**: The boundaries are located on the lattice nodes

Since the boundary handling will be tedious when the boundaries are placed on the lattice nodes, and this is the case for wet nodes, we use **dry nodes** for the implementation.

2.3.1 Bounce-back from objects

The most basic boundary condition is **rigid wall** or the **bounce-back boundary condition**. In this condition, we apply the process without slip condition at the boundary. The equation at the boundary is computed as [12]:

$$f_i(\boldsymbol{x}_b, t + \Delta t) = f_{i^\star}^\star(\boldsymbol{x}_b, t) \tag{2.8}$$

When the **boundary moves** with the velocity of $\boldsymbol{U}_w$, the variation in the momentum of particles must be taken into consideration and the equation is modified as follows [12]:

$$f_i(\boldsymbol{x}_b, t + \Delta t) = f_{i^\star}^\star(\boldsymbol{x}_b, t) - 2 w_i \rho_w \frac{\boldsymbol{c}_i \cdot \boldsymbol{U}_w}{c_s^2} \tag{2.9}$$

where c_s is the speed of sound and ρ_w is the density at the wall. The computation of ρ_w is usually performed by either of the followings [13, 14]:

1. Take the average density $\bar{\rho}$ of the simulated field
2. Extrapolate ρ_w using the particle probability density function in the physical domain

For simplicity, we **take the first solution**.

2.3.2 Periodic boundary conditions (PBC)

In this section, we assume that we have boundaries at $x = 0$ (inlet) and $X - 1$ (outlet) where X is the number of the lattice grid in the x-axis. The most basic PBC assumes that the flow from outlet comes in from inlet as follows [12]:

$$f(0, y, t) = f((X-1)\Delta x, y, t) \tag{2.10}$$

This condition is implicitly implemented during the streaming operation. Another PBC handles the pressure variation Δp between inlet and outlet. Since the density ρ is computed as $\rho = \frac{p}{c_s^2}$ where p is the pressure and c_s is the speed of sound, the density at the inlet ρ_{in} and that at the outlet ρ_{out} can be computed accordingly given the constant pressure p_{out} at the outlet. Then the prestreaming $f^\star$ at the inlet and the outlet are computed as follows [12]:

$$\begin{aligned} f_i^\star(-\Delta x, y, t) &= f_i^{\text{eq}}(\rho_{\text{in}}, \boldsymbol{u}((X-1)\Delta x, y, t)) + (f_i^\star((X-1)\Delta x, y, t) - f_i^{\text{eq}}((X-1)\Delta x, y, t)) \\ f_i^\star(X\Delta x, y, t) &= f_i^{\text{eq}}(\rho_{\text{out}}, \boldsymbol{u}(0, y, t)) + (f_i^\star(0, y, t) - f_i^{\text{eq}}(0, y, t)) \end{aligned} \tag{2.11}$$

where $x = -\Delta x$ and $x = X\Delta x$ correspond to $x = (X-1)\Delta x$ and $x = 0$ in this setting. Note that since the pressure PBC computes the prestreaming $f^\star$, **it must be performed before the streaming operation** unlike the bounce-back.

3

Implementation

In this chapter, we describe how the LBM is implemented in `Python` and how to compute the LBM in parallel. All the implementation is assuming that the physical domain is discretized by D2Q9 and the horizontal axis is x and the vertical axis is y, respectively. Note that entire codes are based on `Numpy` [1] and `mpi4py` [2]. Throughout the chapter, `numpy` is imported as `np`.

3.1 Main routine

Algorithm 1 shows the pseudocode of the main processing in the LBM. Recall that $f(\cdot, t).\text{shape} = (X, Y, 9)$, $\rho(\cdot, 0).\text{shape} = (X, Y)$ and $\boldsymbol{u}(\cdot, 0).\text{shape} = (X, Y, 2)$. First, we provide the initial values for the density and the velocity. Then, we compute the probability function and equilibrium and apply the collision step. The equilibrium implementation is shown in Algorithm 2. After applying equilibrium, we perform the streaming operation shown in Algorithm 3 and slide each quantity to the adjacent cells. Finally, we apply the boundary handling at each boundary cell as described in Algorithm 4 and update the density and the velocity as in Eq (2.5). Note that the order of each step might vary depending on literature [2, 12]. Since `Python` slows down when using for loops and `Python` speeds up when replacing for loops with `numpy` processing, the implementations use as much slicing as possible and high dependency on `numpy` achieves 100 times speed up depending on the settings [15].

Algorithm 3 uses the `np.roll` operation that enables to handle the PBC automatically. This function rolls the array in the following manner:

$$\begin{aligned} \text{np.roll}(f[x][y][i], \text{shift} = \boldsymbol{c}_i, \text{axis} = (0,1)) &= f[nx][ny][i] \\ \text{where } nx = (x + \boldsymbol{c}_i[0]) \% X, ny &= (y + \boldsymbol{c}_i[1]) \% Y \end{aligned} \tag{3.1}$$

where i is the direction index in D2Q9 and $\boldsymbol{c}_i$ is the vector that specifies the i-th direction in D2Q9. In Algorithm 4, we use `in_indices` and `out_indices` to eliminate for-loop by slicing. Additionally, we compute ρ_w by the average density [14]. Note that although the pressure PBC is included in Algorithm 4 for simplicity, only the pressure PBC updates the pre-streaming $f^\star$ and thus we need to perform it **before the streaming operation**. Additioinally, the domain is extended with virtual nodes at both edges of the periodic boundary in the pressure PBC so that we can handle the boundary condition naturally.

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

Algorithm 1 The main routine of the lattice Boltzmann method

```
    The grid size: $X, Y$, Relaxation factor : $\omega$, Initial velocity: $\boldsymbol{u}_0$, Initial density: $\rho_0$   ▷ Inputs
    Boundary conditions
1: function LATTICE BOLTZMANN METHOD
2:     $\rho(\boldsymbol{x}, 0) = \rho_0, \boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0$ for all $\boldsymbol{x} \in [0, X) \times [0, Y)$
3:     for $t = 0, 1, \dots$ do
4:         $f^{\text{eq}}(\cdot, t)$ = equilibrium($\rho(\cdot, t), \boldsymbol{u}(\cdot, t)$)   ▷ Eq (2.7)
5:         $f^\star = f + \omega(f^{\text{eq}} - f)$   ▷ Eq (2.6)
6:         $f^\star(\cdot, t)$ =streaming($f^\star(\cdot, t)$)   ▷ Eq (2.6)
7:         $f(\cdot, t+1)$ = boundary_handling($f^\star(\cdot, t), f^{\text{eq}}(\cdot, t)$)   ▷ Eq (2.8), (2.9), (2.11)
8:         $\rho(\cdot, t+1), \boldsymbol{u}(\cdot, t+1)$=moments_update($f(\cdot, t+1)$)   ▷ Eq (2.5)
```

Algorithm 2 equilibrium

```
    $\boldsymbol{w}$ = np.array([$\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}$]), $\boldsymbol{c}$ in Eq (2.4)
1: function EQUILIBRIUM($\rho = \rho(\cdot, t)$, $\boldsymbol{u} = \boldsymbol{u}(\cdot, t)$)   ▷ $\boldsymbol{u}$.shape = $(X, Y, 2)$, $\rho$.shape = $(X, Y)$
2:     u_norm2 = ($\boldsymbol{u}$ ** 2).sum(axis=-1)[..., None]
3:     u_at_c = $\boldsymbol{u}$ @ $\boldsymbol{c}^\top$   ▷ u_at_c.shape = $(X, Y, 9)$
4:     w_tmp, $\rho$_tmp = $\boldsymbol{w}$[None, None, ...], $\rho$[..., None]   ▷ Adapt the shapes to u_at_c
5:     $f^{\text{eq}}$ = w_tmp * $\rho$_tmp * (1 + 3 * u_at_c + 4.5 * (u_at_c) ** 2)-1.5 * u_norm2
6:     return $f^{\text{eq}}$
```

3.2 Parallel computation by MPI

In order to process the LBM in parallel, we employ the spatial domain decomposition and the messaging passing interface (MPI) so that we can compute the collision step of the LBM in parallel. This is possible because the collision step does not require any communication between processes [16]. Then, we explain how we divide the domain. Suppose we are provided the number of processes of P, we first factorize P such that $P = P_x \times P_y$ where $P_x, P_y \in \mathbb{Z}^+$, $P_x, P_y = \arg\min_{P_x, P_y}(||P_y - P_x||)$ and $P_x \leq P_y$ if $X \leq Y$ otherwise $P_y \leq P_x$. Then, we divide the x-axis into P_x intervals and the y-axis into P_y intervals where any pairs of intervals I_i, I_j in the same direction satisfy $-1 \leq ||I_i|| - ||I_j|| \leq 1$. Note that $||I||$ is the size of the interval I. This split of the domain achieves the most balanced distribution of the computation. For the streaming step, we need to consider particles moving from one process to another. We implement it using so-called **ghost cells** around the actual computational domain. Figure 3.1 shows the conceptual visualization of how each process communicates and ghost cells work. Since each process requires the four edges of adjacent processes, the communications are required four

Algorithm 3 Streaming operation

```
    $\boldsymbol{c}$ in Eq (2.4)
1: function STREAMING($f^\star = f^\star(\cdot, t)$)
2:     $f^{\text{post}}$ = np.zeros_like($f^\star$)
3:     for $i = 0, 1, \dots, 8$ do
4:         $f^{\text{post}}$[..., $i$]=np.roll($f^\star$[..., i], shift=$\boldsymbol{c}_i$, axis=(0, 1))   ▷ Slide $f^\star$ one step to c[i]
5:     return $f^{\text{post}}$
```

Algorithm 4 Boundary conditions (Pressure PBC is also included for simplicity)

```
    The indices in D2Q9 s.t. the flow comes in given boundaries: in_indices
    The indices in D2Q9 s.t. the flow goes out given boundaries: out_indices
 1: function BOUNDARY HANDLLING($f^\star = f^\star(\cdot, t)$, $f^{\mathrm{eq}} = f^{\mathrm{eq}}(\cdot, t)$)
 2:     if Pressure PBC then                                  ▷ fluid flows from $x = 0$ to $X - 1$
 3:         # Note: Pressure PBC must be applied before streaming operation
 4:         $f^{\mathrm{eq}}_{\mathrm{in}}, f^{\mathrm{eq}}_{\mathrm{out}}$ = equilibrium($\rho_{\mathrm{in}}$, $\boldsymbol{u}$[-2]), equilibrium($\rho_{\mathrm{out}}$, $\boldsymbol{u}$[1])
 5:         $f^\star$[0, :,out_indices]=$f^{\mathrm{eq}}_{\mathrm{in}}$[:,out_indices].T+($f^\star$[-2, :,out_indices]-$f^{\mathrm{eq}}$[-2, :,out_indices])
 6:         $f^\star$[-1, :,in_indices]=$f^{\mathrm{eq}}_{\mathrm{out}}$[:,in_indices].T+($f^\star$[1, :,in_indices] - $f^{\mathrm{eq}}$[1, :,in_indices])
 7:     if Rigid wall then                                    ▷ The case when the wall is at the top
 8:         $f$[:, -1, in_indices] = $f^\star$[:, -1, out_indices]
 9:     if Moving wall then                                   ▷ The case when the wall is at the top
10:         coef = np.zeros_like((X, Y, 9))
11:         value = 2 * $\boldsymbol{w}$[out_indices] * ($\boldsymbol{c}$[out_indices] @ $\boldsymbol{u}$) / $c_s$ ** 2
12:         coef[:, -1, out_indices] = value[np.newaxis, :]
13:         $f$[:, -1, in_indices] = $f^\star$[:, -1, out_indices] $-\rho_w$ * coef[:, -1, out_indices]
14:     return $f$
```

times for each process. Algorithm 5 shows the implementation using `mpi4py`. `grid_manager` is the self-developed module that manages useful information related to the process location, the adjacent relation, and so on. `Sendrecv` function is used for the communication and each process receives an array from `dest` that is sent by `neighbor` and sends an array `sendbuf` to `neighbor`. Note that `buf` is the abbreviation of buffer and used for the buffer to communicate data.

3.3 Software quality

All the codes follow `pep8 style` [3] and `Google Python Style documentation string` [4]. In order to make the codes robust to unexpected errors, we introduce `Flake8` [5] and `MyPy` static typing check [6] as well. Furthermore, all the components are tested by `unittest`[7] and we provide `requirements.txt` and the shell scripts for the main experiments to reproduce the complete running conditions. Those tools **guarantee the reproducibility** of the experiments. Furthermore, the implementations focus on abstraction and most codes are abstracted to reduce the coding lines as much as possible. Therefore, the codes are **highly reusable** and the implementation has only one explicit coding for each Algorithm provided in this chapter. Furthermore, `ArgumentParser` allows users to pass an arbitrary setting to run the experiments and it **contributes to the generality** in this code. All the instructions are available at `Github` repository described in Chapter 1.

[3]https://www.python.org/dev/peps/pep-0008/
[4]https://google.github.io/styleguide/pyguide.html
[5]https://flake8.pycqa.org/en/latest/
[6]http://mypy-lang.org/
[7]https://docs.python.org/3/library/unittest.html

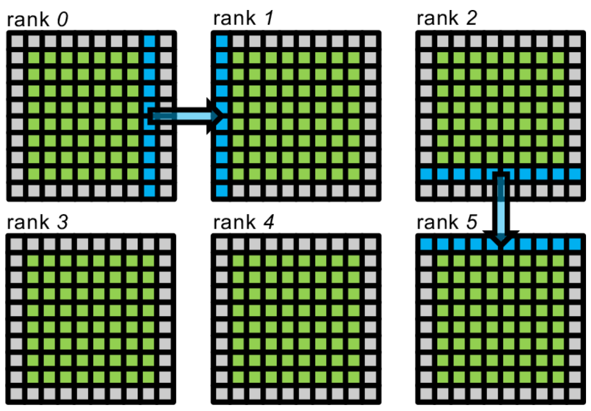


Figure 3.1: Domain decomposition and communication strategy in MPI. As described in the main text, we first divide each axis into P_x and P_y intervals and divide by the intervals. Each rank has green lattice points and this area is the active physical domain. Then we add additional ghost cells for buffer (gray lattice points). During each communication step, the outermost green active lattice sends the data to the adjacent outermost ghost lattice (blue arrows). The figure is cited from Figure 2 in [16].

Algorithm 5 The communication of the particle probability density function

```
    Process and lattice grids management: grid_manager
 1: function COMMUNICATION
 2:     for dir in grid_manager.neighbor_directions do          ▷ Iterate over the D2Q9 index
 3:         dx, dy = $\boldsymbol{c}_i$
 4:         sendidx = grid_manager.step_to_idx(dx, dy, send=True)
 5:         recvidx = grid_manager.step_to_idx(dx, dy, send=False)
 6:         neighbor = grid_manager.get_neighbor_rank(dir)
 7:         if dx == 0 then                                      ▷ send to top and bottom
 8:             sendbuf = $f$[:, sendidx, ...].copy()
 9:             grid_manager.rank_grid.Sendrecv(sendbuf=sendbuf, dest=neighbor,
10:                                             recvbuf=recvbuf, source=neighbor)
11:             $f$[:, recvidx, ...] = recvbuf
12:         else if dy == 0 then                                 ▷ send to left and right
13:             sendbuf = $f$[sendidx, ...].copy()
14:             grid_manager.rank_grid.Sendrecv(sendbuf=sendbuf, dest=neighbor,
15:                                             recvbuf=recvbuf, source=neighbor)
16:             $f$[recvidx, ...] = recvbuf
17:     return $f$
```

4

Numerical results

In the previous chapter, we discuss the implementation details and how we apply LBM to various settings. In this chapter, we first illustrate how to validate the implementations and then show the visualizations and numerical results obtained from the series experiments.

4.1 Validation experiments

In the physics simulation, it is always important to validate whether the implementations are correct. Therefore, we first show how to validate the implementation using several examples.

4.1.1 Shear wave decay

The shear wave decay represents the time evolution of a velocity perturbation in the flow. Since the viscosity decays the velocity of the flow, the velocity converges to zero in the end. When we set the following sinusoidal perturbation in the velocity as the initial condition:

$$\boldsymbol{u}(\boldsymbol{x}, t=0) = \begin{bmatrix} u_x(y, t=0) \\ 0 \end{bmatrix} = \begin{bmatrix} \epsilon \sin \frac{2\pi y}{Y} \\ 0 \end{bmatrix} \tag{4.1}$$

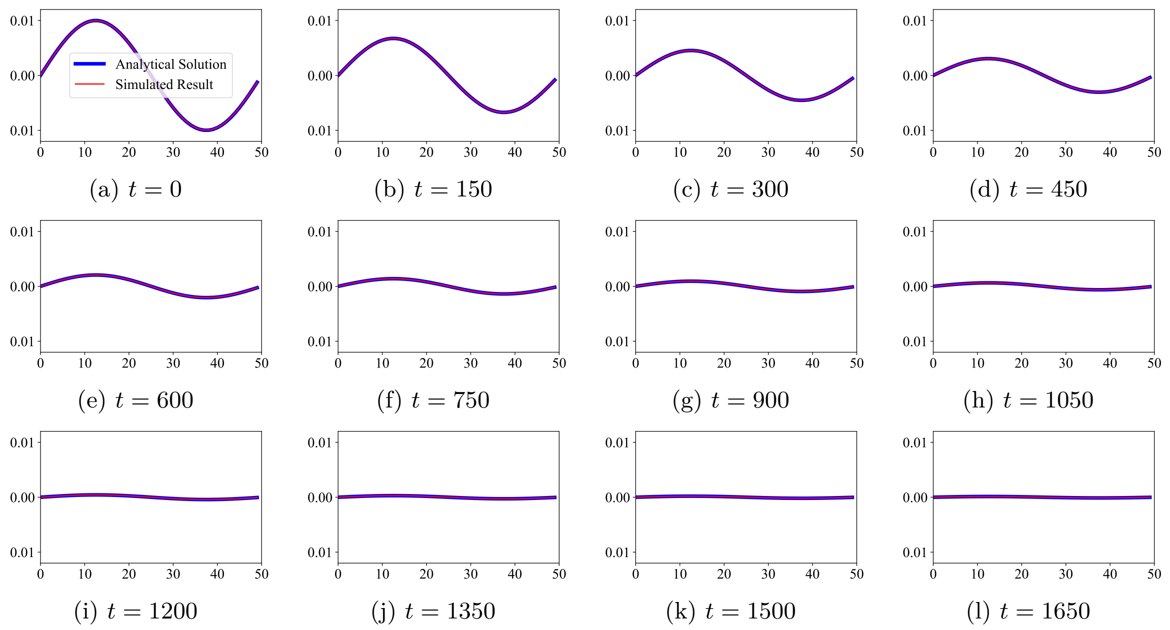
Then the analytical solution for the time evolution of the velocity is calculated as follows [17]:

$$u_x(y, t) = \epsilon \exp\left(-\nu \left(\frac{2\pi}{Y}\right)^2 t\right) \sin \frac{2\pi y}{Y} \tag{4.2}$$

Note that this result is obtained using Navier-Stokes equations for incompressible fluid and the assumptions that the pressure term ∇p and the convection term $(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}$ are negligible compared to the viscosity term $\nu \nabla^2 \boldsymbol{u}$. In Figure 4.1, we show the plot of both simulated results and the analytical solutions of sinusoidal velocity. Note that the initial condition follows Eq (4.1). As seen in the figure, the simulated results and the analytical solutions **perfectly fit** and thus we could validate our implementation of rigid wall and moments updates. Figure 4.2 shows the density distribution over time. This simulation uses the sinusoidal density in the x-direction:

$$\rho(\boldsymbol{x}, 0) = \rho_0 + \epsilon \sin \frac{2\pi x}{X} \tag{4.3}$$

As seen in the figure, the sinusoidal density also yields the convergence. On the other hand, the sinusoidal density has a swing of the maxima and the minima unlike the sinusoidal velocity.



(a) $t = 0$ (b) $t = 150$ (c) $t = 300$ (d) $t = 450$

(e) $t = 600$ (f) $t = 750$ (g) $t = 900$ (h) $t = 1050$

(i) $t = 1200$ (j) $t = 1350$ (k) $t = 1500$ (l) $t = 1650$

Figure 4.1: The time evolution of the sinusoidal velocity (Eq (4.1)) at the $x = 25$ in the lattice grid size of $(50, 50)$. The x-axis shows the location in the y direction and the y-axis shows the magnitude of velocity at the corresponding location. The coefficients ϵ in Eq (4.1) and the initial density ρ_0 are set to 0.01 and 1.0 respectively. The relaxation term ω is set to 1.0.

As discussed, momentum fluctuations decay exponentially and such a decay is represented as follows [18, 19]:

$$Q_t(t) = \exp\left(-\nu\left(\frac{2\pi}{X}\right)^2 t\right) \tag{4.4}$$

where $Q(x,t) = \epsilon Q_x(x)Q_t(t)$ and $Q(x,t)$ is one of the moment quantities. Note that we assume that the assumptions for Eq (4.2) hold and the case of the sinusoidal velocity is equivalent to Eq (4.2) [17]. We perform the experiments to validate the implementation via the viscosity estimated by Eq (4.4) using the exact experiment settings for Figure 4.1 and Figure 4.2 except the relaxation term ω. The analytical viscosity is computed as $\nu = \frac{1}{3}(\frac{1}{\omega} - \frac{1}{2})$. For the experiments, the simulated viscosity is computed based on the exponential decay curve, i.e. Eq (4.4), of the density and velocity using `curve_fit` [1]. `curve_fit` approximates the optimal viscosity ν from the observations. Since the densities swing so much and a smooth exponential decay curve is not obtained, we only take the maximum of the swinging. Such time-series data is obtained by `argrelextrema` [2]. The results are shown in Figure 4.3. Based on the results, ω close to 0.0 and 2.0 leads to numerical instability. Otherwise, the simulated results and analytical solution fit perfectly. Therefore, we need to avoid using ω closer to 0 or 2 for more accurate results.

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html
[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.argrelextrema.html

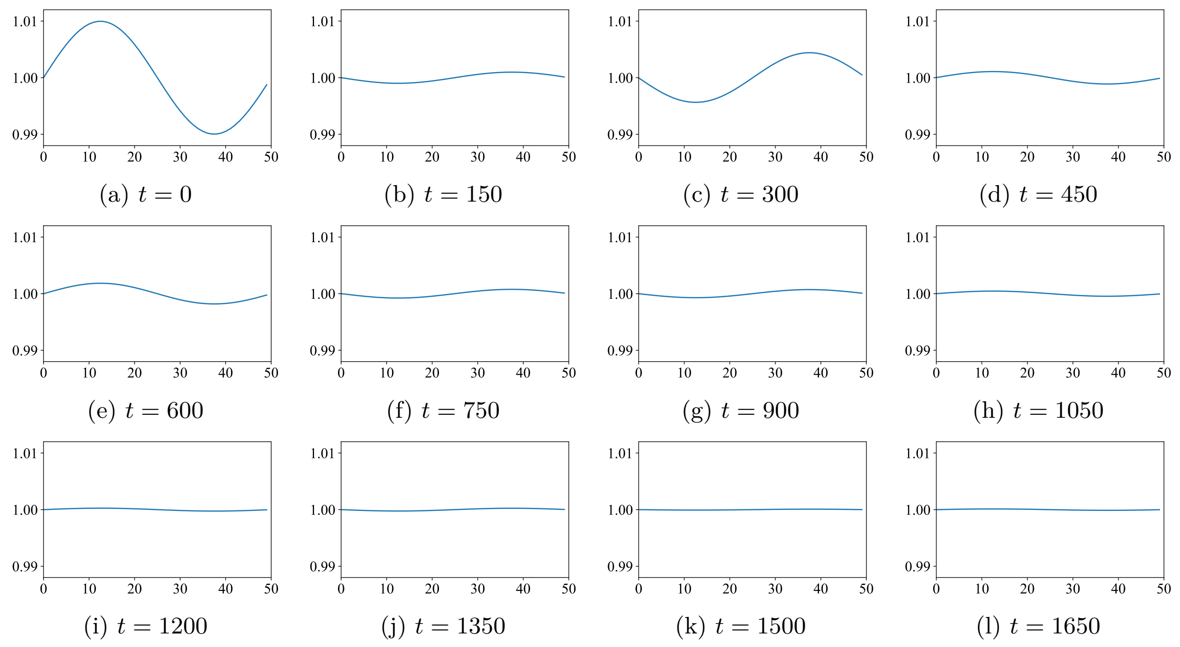


Figure 4.2: The time evolution of the sinusoidal density (Eq (4.3)) at $y = 25$ in the lattice grid size of $(50, 50)$. The x-axis shows the location in the x direction and the y-axis shows the magnitude of density. The coefficients ϵ and ρ_0 in Eq (4.3) are set to 0.01 and 1.0 and the velocity is initialized by $\boldsymbol{u}(\boldsymbol{x}, 0) = (0, 0)$. The relaxation term ω is set to 1.0.

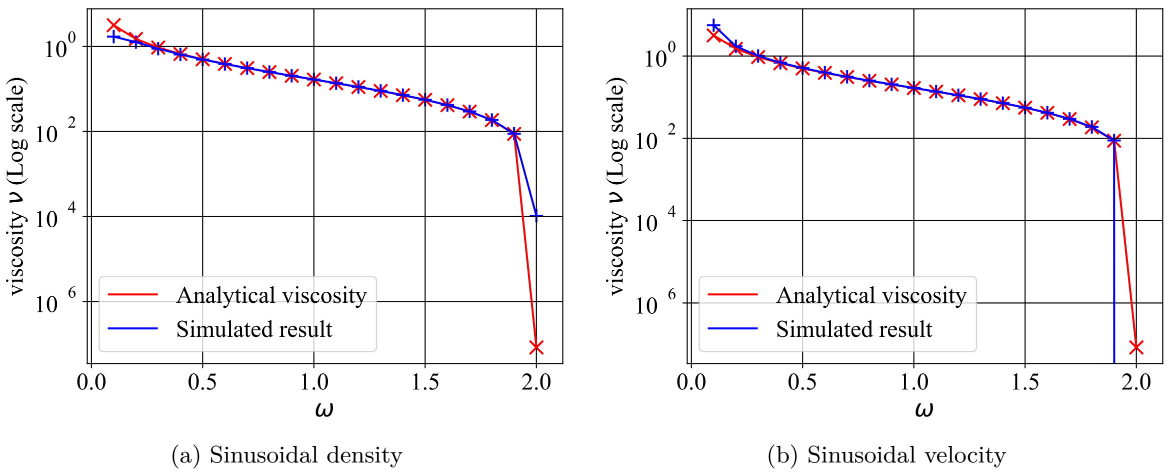


Figure 4.3: The simulated viscosity value over various relaxation values ω. The analytical solution uses $\nu = \frac{1}{3}(\frac{1}{\omega} - \frac{1}{2})$ and the simulated viscosity ν is approximated from an exponential decay curve in Eq (4.4). The simulation is performed $T = 3000$ steps and we take the maximum magnitude of $||u_x||$ for (a) and $||\rho - \rho_0||$ for (b) to fit the curve. Note that (a) uses the same parameters as in Figure 4.1 and (b) uses the same parameters as in Figure 4.2.

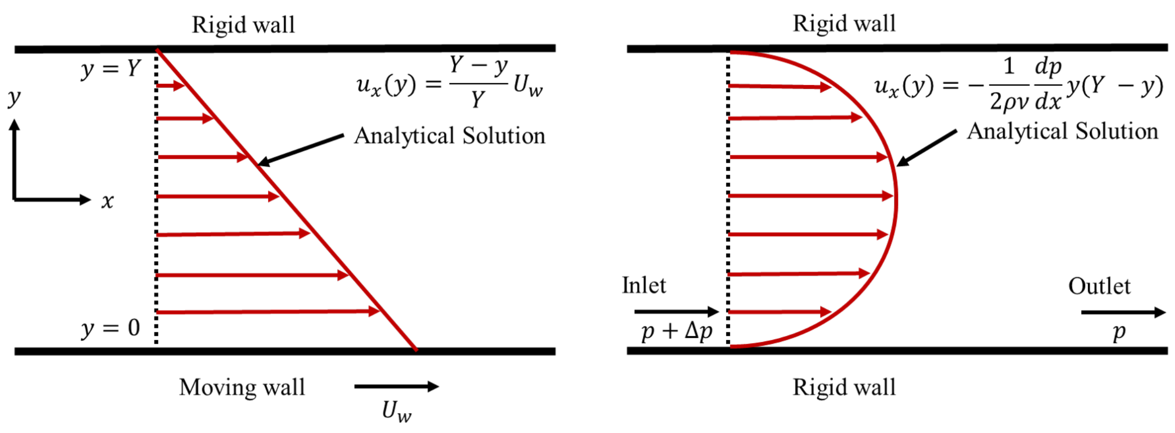


Figure 4.4: The conceptual visualizations of the Couette flow (Left) and Poiseuille flow (Right).

4.1.2 Couette flow

The Couette flow is the flow between two walls as shown in Figure 4.4: One is fixed and the other moves horizontally with the velocity of U_w. The flow is caused by the viscous drag force acting on the fluid. Since the Couette flow also has an analytical solution, we can validate the implementation of the moving wall. The analytical solution for Figure 4.4 is given by [20]:

$$u_x(y) = \frac{Y - y}{Y} U_w \tag{4.5}$$

where Y is the distance between the two walls and $u_x(y)$ is the horizontal velocity of the flow at the location of y. In the experiment, we apply the bounce-back boundary condition at the moving wall and the rigid wall and the PBC at the inlet and outlet. The results are shown in Figure 4.5. As shown in the figures, the flow velocity iteratively approaches the analytical solution and it perfectly fits in the end and **the velocity stops growing** as seen at $t = 9000 \sim 15000$. From this experiment, the moving wall can be validated.

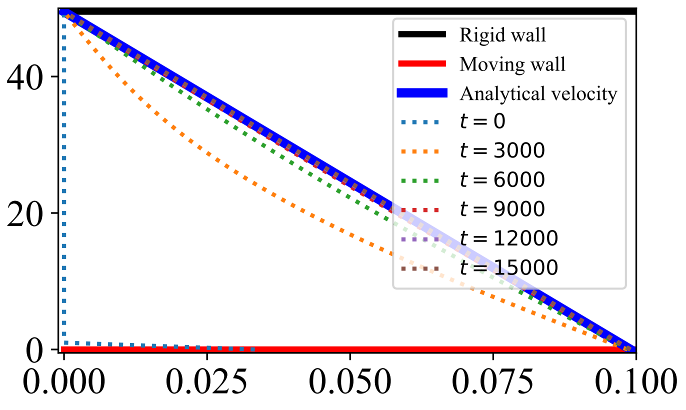


Figure 4.5: The velocity evolution at $x = 25$ in the lattice grid size of $(50, 50)$. The wall velocity U_w at the bottom and the relaxation term ω are set to 0.1 and 1.2 respectively. We use **dry node** as described in Section 2.3 and the computation of wall density follows Section 2.3.1. The initial density and velocity are $\rho(\boldsymbol{x}) = 1.0, \boldsymbol{u}(\boldsymbol{x}) = (0, 0)$.

4.1.3 Poiseuille flow

The Poiseuille flow is the flow between two non-moving walls as shown in Figure 4.4. The flow is caused by a constant pressure difference $\frac{dp}{dx}$ in the horizontal direction of the two walls. The Poiseuille flow also has the analytical solution and we can validate the implementation of the pressure PBC. The analytical solution for Figure 4.4 is given by [21]:

$$u_x(y) = -\frac{1}{2\rho\nu}\frac{dp}{dx}y(Y-y) \tag{4.6}$$

In the experiment, we apply the bounce-back boundary condition at the moving wall and the rigid wall and the pressure PBC at the inlet and outlet. Figure 4.6 presents the results and the simulated results approach the analytical solutions as in the Couette flow. In the end, it fits completely and **the velocity stops growing** as seen at $t = 12000 \sim 18000$.

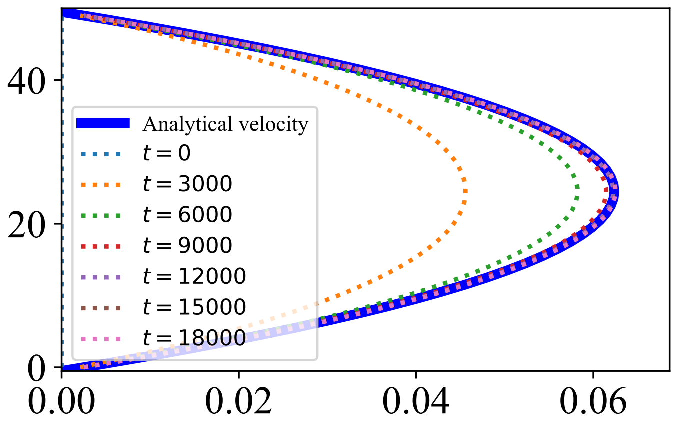


Figure 4.6: The velocity evolution at $x = 25$ in the lattice grid size of $(50, 50)$. The relaxation term ω is set to 1.2. The density at the inlet ρ_{in} and the density at the outlet ρ_{out} are set to 0.301 and 0.3 respectively. The initial density and velocity are $\rho(\boldsymbol{x}) = 1.0, \boldsymbol{u}(\boldsymbol{x}) = (0, 0)$.

4.2 Lid-driven cavity

Finally, we handle a concrete example. In this paper, the lid-driven cavity shown in Figure 4.7 is simulated. The lid-driven cavity simulates the flow inside a box with three rigid walls and one moving wall, i.e. a lid. In this simulation, the turbulence is caused when the following Reynolds number is larger than 1000 [22]:

$$\text{Re} = \frac{LU}{\nu} \tag{4.7}$$

where L is the characteristic length parameter of the body and U is the stream flow velocity. One key property of the Reynolds number is that two flow system is dynamically similar if the Reynolds number and the geometry are similar [23]. Therefore, we present the results with various viscosity ν and the wall velocity $U = U_w$ that satisfy the Reynolds number of 1000 under $L = X = Y = 300$ in Figure 4.8. In the figures, all the settings converge to a similar flow in the end as indicated in the key property of the Reynolds number. Figure 4.9 shows the time evolution of the streaming plot with the Reynolds number of 1000. The series of figure

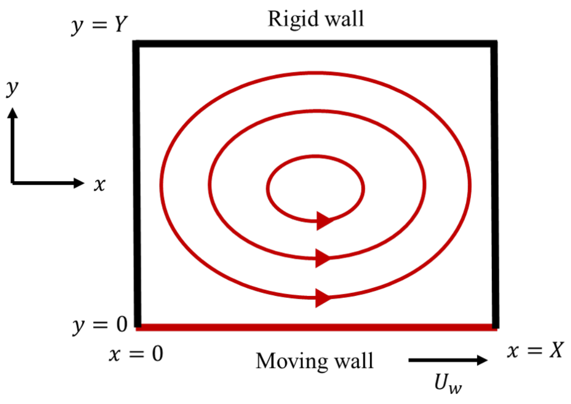


Figure 4.7: The conceptual visualizations of the lid-driven cavity.

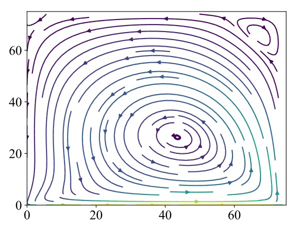
Table 4.1: The validation of the parallel implementation by comparing the velocity field in the serial and the parallel implementations. The parallel implementation is performed by the number of processes $P = 9$. We set the lattice grid size is $(X, Y) = (30, 30)$, the wall velocity $U_w = 0.1$ and the viscosity $\nu = 0.03$ and perform $T = 10000$ updates. The initial density and velocity are $\rho(\boldsymbol{x}) = 1.0, \boldsymbol{u}(\boldsymbol{x}) = (0, 0)$.

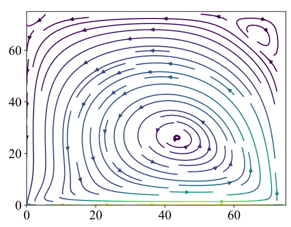
	Min	Max	Sum of absolute values
Velocity $\boldsymbol{u}_p$ in parallel implementation	-0.03488	0.08998	20.12832
Velocity $\boldsymbol{u}_s$ in serial implementation	-0.03488	0.08998	20.12832
The absolute difference $\|\|\boldsymbol{u}_p - \boldsymbol{u}_s\|\|$	**0.0**	**0.0**	**0.0**

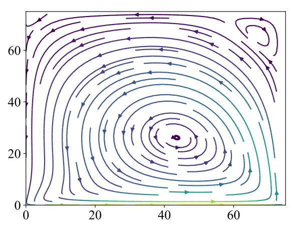
shows that the streaming changes gradually and starts to have spirals at a corner due to the turbulence. **The time evolution of the velocity streaming plot is provided in Github** [3]. Note that all the experiments for Figure 4.8, 4.9 are **performed using MPI of 9 processes** and Table 4.1 shows the validation of the parallel implementation. Since the summation of the absolute error of the velocity over the whole domain is 0.0, it is obvious that **the parallel implementation behaves identically to the serial implementation**.

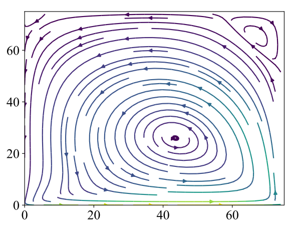
This experiment requires a long time to complete. For example, it takes 1 hour to finish one simulation using intel core i7–10700 and 32GB RAM. Recall that the advantage of the LBM is to allow us to compute the simulation in parallel easily. For this reason, we test the scalability of this simulation using various numbers of processes. Note that all the experiments related to the scaling test are performed on **BWUniCluster** [4]. The implementation follows Section 3.2 and each thread is bound to one processor. Figure 4.10 shows the plot of MLUPS, a.k.a. million lattice updates per second, and the number of processes. As seen in the figure, the larger grid size leads to less MLUPS with the smaller number of processes. This is due to the heavy load on small number of processors. On the other hand, as the number of processes becomes larger, the simulation with a larger domain exhibits higher efficiency. Ideally, the MLUPS should grow linearly with respect to the number of processes. However, it does not happen because of the latency of the communication and the waiting for the synchronization as described in Amdahl's law [24]. It explains why the larger domain leads to more scalability with respect to the number of processes.

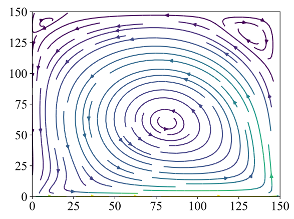
[3]https://github.com/nabenabe0928/high-performance-computing-fluid-dynamics-with-python/
[4]https://wiki.bwhpc.de/e/Category:BwUniCluster_2.0

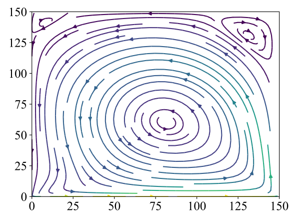

(a) $U_w = 0.1, \nu = 0.03$

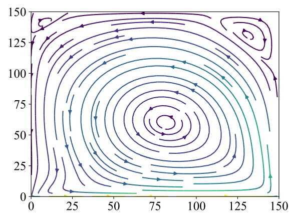

(b) $U_w = 0.2, \nu = 0.06$

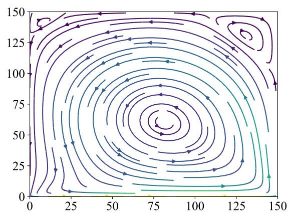

(c) $U_w = 0.3, \nu = 0.09$

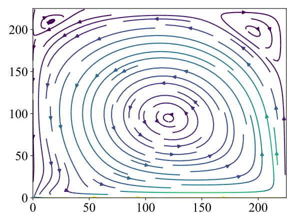

(d) $U_w = 0.4, \nu = 0.12$

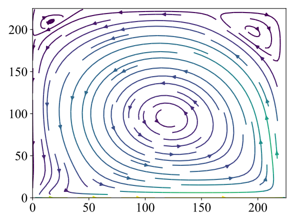

(e) $U_w = 0.1, \nu = 0.03$

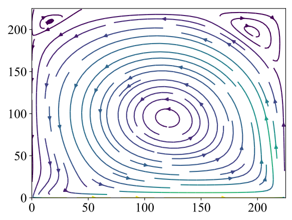

(f) $U_w = 0.2, \nu = 0.06$

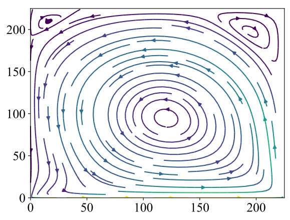

(g) $U_w = 0.3, \nu = 0.09$

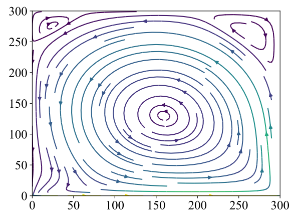

(h) $U_w = 0.4, \nu = 0.12$

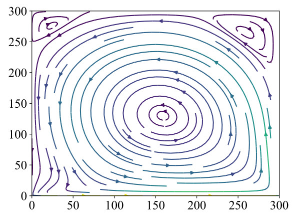

(i) $U_w = 0.1, \nu = 0.03$

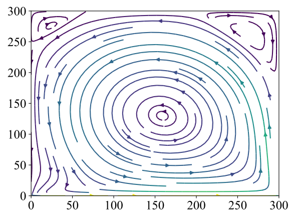

(j) $U_w = 0.2, \nu = 0.06$


(k) $U_w = 0.3, \nu = 0.09$


(l) $U_w = 0.4, \nu = 0.12$


(m) $U_w = 0.1, \nu = 0.03$


(n) $U_w = 0.2, \nu = 0.06$


(o) $U_w = 0.3, \nu = 0.09$

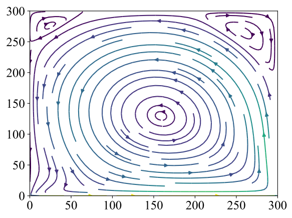

(p) $U_w = 0.4, \nu = 0.12$

Figure 4.8: The stream plots of the lid-driven cavity with the lattice grid size of $(75, 75), (150, 150), (225, 225), (300, 300)$. The setting of the wall follows Figure 4.7. (a) – (d), (e) – (h), (i) – (l), (m) – (p) are chosen to satisfy the Reynolds number 250, 500, 750, 1000, respectively. We perform the update $T = 100000$ times for each setting. The computation of wall density follows Section 2.3.1. The initial density and velocity are $\rho(\boldsymbol{x}) = 1.0, \boldsymbol{u}(\boldsymbol{x}) = (0, 0)$.

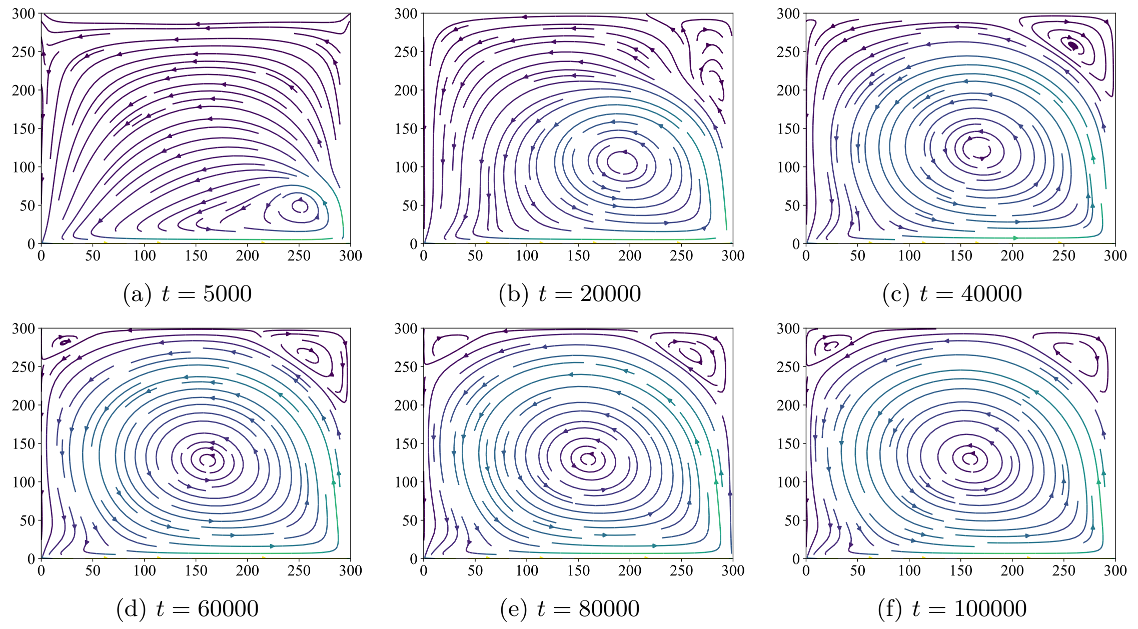

(a) $t = 5000$ (b) $t = 20000$ (c) $t = 40000$ (d) $t = 60000$ (e) $t = 80000$ (f) $t = 100000$

Figure 4.9: The time evolution of the stream plots of the lid-driven cavity with the Reynolds number of 1000. The setting of the wall follows Figure 4.7. In this experiment, the lattice grid size is $(300, 300)$, the viscosity ν and the wall velocity are set to 0.03 and 0.1, respectively. The computation of wall density follows Section 2.3.1. The initial density and velocity are $\rho(\boldsymbol{x}) = 1.0, \boldsymbol{u}(\boldsymbol{x}) = (0, 0)$. **The gif file for this experiment is available at Github** as described in footnote 3.

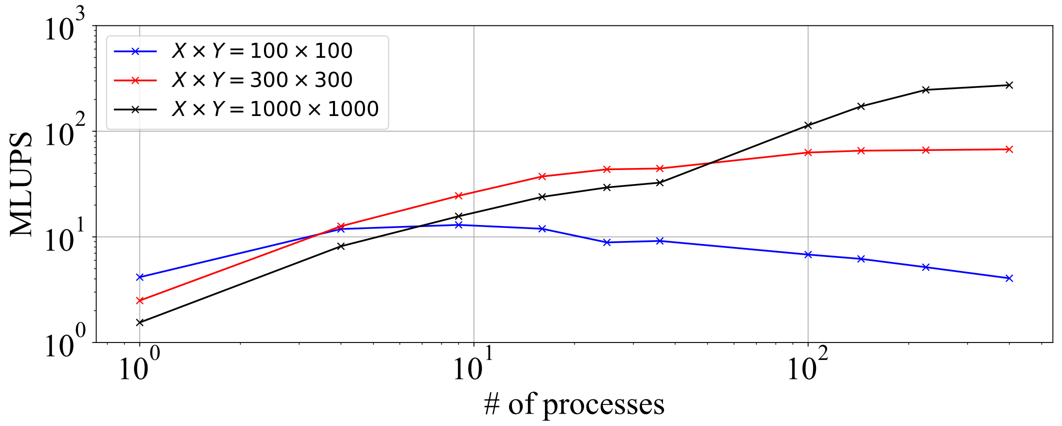


Figure 4.10: The scaling test of the lid-driven cavity simulation. The grid size is either $100 \times 100, 300 \times 300$ or 1000×1000. The number of processes are $1, 4, 9, 16, 25, 36, 100, 144, 225, 400$ respectively. Note that both axes are log-scale. The viscosity and the wall velocity are set to $\nu = 0.03$ and 0.1 and we perform the update $T = 10000$ times. The initial density and velocity are $\rho(\boldsymbol{x}) = 1.0, \boldsymbol{u}(\boldsymbol{x}) = (0, 0)$.

5

Conclusions

In this paper, we describe what the LBM is and how we implement it. Chapter 1 describes the motivation behind the numerical integrations and mentions the advantages of the LBM, i.e. simple implementation and scalability with respect to the computational resources.

Chapter 2 explains the theoretical aspects of LBM and how those equations are plugged into the computational simulations. More specifically, the discretization of each equation and the boundary handlings are presented.

Chapter 3 shows the algorithms of each component. Especially, we focus on the descriptions how the simulation should be implemented using numpy which is effective to speed up `Python` implementations. Additionally, the MPI usage and the domain division method are described. Note that each algorithm in our implementation is tested using `unittest` and abstracted as much as possible so that each component can be reused and we can reduce the bugs over the whole implementation.

Chapter 4 presents the validations of each component using the comparison between the analytical solutions and visualizes how the LBM works in the lid-driven cavity example. For the validations, we use the shear wave decay, the Couette flow and the Poissuille flow and show the analytical solutions for each phenomenon. After the validations, we show the lid-driven cavity simulation exhibits similar dynamics when we have a constant Reynolds number. Furthermore, we test the scalability of the LBM in the lid-driven cavity simulation. The experiments show that the 30 times speedup using 100 processes for 300×300 grids and bad scalability in the smaller grids' settings. This observation corresponds to the intuition from Amdahl's law.

Bibliography

[1] Praveen Padagannavar and Manohara Bheemanna. Automotive computational fluid dynamics simulation of a car using ansys. International Journal of Mechanical Engineering and Technology (IJMET) Volume, 7:91–104, 2016.

[2] Krüger Timm, H Kusumaatmaja, A Kuzmin, O Shardt, G Silva, and E Viggen. The lattice Boltzmann method: principles and practice. Springer: Berlin, Germany, 2016.

[3] Kerson Huang. Statistical mechanics, john wily & sons. New York, page 10, 1963.

[4] D Raabe. Overview of the lattice boltzmann method for nano-and microscale fluid dynamics in materials science and engineering. Modelling and Simulation in Materials Science and Engineering, 12(6):R13, 2004.

[5] Guy R McNamara and Gianluigi Zanetti. Use of the boltzmann equation to simulate lattice-gas automata. Physical review letters, 61(20):2332, 1988.

[6] Prabhu Lal Bhatnagar, Eugene P Gross, and Max Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. Physical review, 94(3):511, 1954.

[7] B Caroli, C Caroli, and B Roulet. Non-equilibrium thermodynamics of the solidification problem. Journal of crystal growth, 66(3):575–585, 1984.

[8] Roger Peyret and Thomas D Taylor. Computational methods for fluid flow.

[9] James D Sterling and Shiyi Chen. Stability analysis of lattice boltzmann methods. Journal of Computational Physics, 123(1):196–206, 1996.

[10] Guo Zhao-Li, Zheng Chu-Guang, and Shi Bao-Chang. Non-equilibrium extrapolation method for velocity and pressure boundary conditions in the lattice boltzmann method. Chinese Physics, 11(4):366, 2002.

[11] H Liu and JG Zhou. Lattice boltzmann approach to simulating a wetting–drying front in shallow flows. Journal of fluid mechanics, 743:32–59, 2014.

[12] Sauro Succi. The lattice Boltzmann equation: for complex states of flowing matter. Oxford University Press, 2018.

[13] Qisu Zou and Xiaoyi He. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. Physics of fluids, 9(6):1591–1598, 1997.

[14] Sorush Khajepor, Jing Cui, Marius Dewar, and Baixin Chen. A study of wall boundary conditions in pseudopotential lattice boltzmann models. Computers & Fluids, 193:103896, 2019.

[15] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. Computing in science & engineering, 13(2):22–30, 2011.

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

[17] Linlin Fei, Kai H Luo, and Qing Li. Three-dimensional cascaded lattice boltzmann method: Improved implementation and consistent forcing scheme. Physical Review E, 97(5):053309, 2018.

[18] Bruce J Palmer. Transverse-current autocorrelation-function calculations of the shear viscosity for molecular liquids. Physical Review E, 49(1):359, 1994.

[19] Berk Hess. Determining the shear viscosity of model liquids from molecular dynamics simulations. The Journal of chemical physics, 116(1):209–217, 2002.

[20] Péter Nagy-György and Csaba Hős. A graphical technique for solving the couette-poiseuille problem for generalized newtonian fluids. Periodica Polytechnica Chemical Engineering, 63(1):200–209, 2019.

[21] AA Mendiburu, LR Carrocci, and JA Carvalho. Analytical solution for transient onedimensional couette flow considering constant and time-dependent pressure gradients. Revista de Engenharia Térmica, 8(2):92–98, 2009.

[22] TP Chiang, WH Sheu, and Robert R Hwang. Effect of reynolds number on the eddy structure in a lid-driven cavity. International journal for numerical methods in fluids, 26(5):557–579, 1998.

[23] Pijush K Kundu, Ira M Cohen, and D Dowling. Fluid mechanics 4th, 2008.

[24] Gene 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, pages 483–485, 1967.