High Performance Computing with Python

Final Report

Rushang Phira

5578541

phirar@informatik.uni-freiburg.de

August 14, 2023

Contents

1 Introduction **2**
- 1.1 Setup . . . 2
- 1.2 Software and Hardware . . . 2

2 Methods **4**
- 2.1 Streaming . . . 4
- 2.2 Collision . . . 4
- 2.3 Boundary Conditions . . . 5
  - 2.3.1 Periodic boundary . . . 5
  - 2.3.2 Periodic boundary with pressure gradient . . . 6
  - 2.3.3 Rigid wall . . . 6
  - 2.3.4 Moving wall . . . 7

3 Implementation **8**
- 3.1 Streaming . . . 8
- 3.2 Collision . . . 9
- 3.3 Boundary Handling . . . 9
- 3.4 Parallellization . . . 11

4 Experiments and Results **13**
- 4.1 Shear Wave Decay . . . 13
  - 4.1.1 Density . . . 14
  - 4.1.2 Velocity . . . 14
- 4.2 Couette Flow . . . 16
- 4.3 Poiseuille Flow . . . 17
- 4.4 Sliding Lid . . . 17
- 4.5 Notes on Reproducibility . . . 20
  - 4.5.1 Local, Serial . . . 20
  - 4.5.2 Local, Parallel . . . 20
  - 4.5.3 BwUniCluster . . . 20

5 Conclusion **23**

1

Introduction

This report is structured around the "High-Performance Computing: Fluid Mechanics with Python" course offered by the University of Freiburg. We aim to implement and parallelize the Lattice Boltzmann Method in this course. The lattice Boltzmann models (LBM) are solutions for the numerical solution of (nonlinear) partial differential equations [1]. According to Bao and Meskas, the LBM can be viewed as a finite difference method for solving the Boltzmann transport equation. Moreover, the Navier-Stokes equations can be recovered by the LBM with a proper choice of the collision operator [2]. Boltzmann's kinetic theory of gases formed the base for the Boltzmann transport equation, which is discretized by the LBM. According to Fei et al., "the LBM solves a discrete Boltzmann equation which is designed to reproduce the Navier-Stokes (N-S) equations in the macroscopic limit" [3]. We use this discretized version in two dimensions for our implementation.

The primary method of implementation involves simulating the flow of a fluid in a two-dimensional lattice. We perform streaming and collision operations, among others, and instead of using the Navier-Stokes equations, we compare the analytical solutions with the empirical results we obtain through our experiments [4].

1.1 Setup

In the LBM, we divide the grid into lattices where each lattice node represents a particle. Each of these particles stream along specified directions to the neighboring nodes. The number of directions and linkage depends on the lattice arrangement [5]. The lattice arrangement used here is the D2Q9 which represents a discretized two-dimensional space with nine channels (one for each direction) as shown in figure 2.1. Here, the distance between each adjacent grid point signifies the distance traversed by a particle for a time step Δt.

1.2 Software and Hardware

Most experiments are run locally, on a system with an AMD Ryzen 5 3550H (4 cores, 4 threads) CPU with 8 gigabytes of DDR4 single-channel memory running at 2666 MHz.

The experiments are performed using Python 3.11.4 locally. The packages mainly used are `numpy` and `matplotlib`. For parallelization, `ipyparallel` and `mpi4py` are used. Part of the sliding lid experiment is run on BWUniCluster 2.0, within which Python 3.8.6 with GNU 10.2 compiler is used. `openMPI` is used for parallelization. To run jobs, `dev_multiple`, `dev_multiple_il`, `multiple`, and `multiple_il` channels are used.

2

Methods

2.1 Streaming

The Boltzmann transport equation is written as:

$$\frac{\delta f(\mathbf{r}, \mathbf{v}, \mathbf{t})}{\delta t} + v\Delta r f(\mathbf{r}, \mathbf{v}, \mathbf{t}) + a f(\mathbf{r}, \mathbf{v}, \mathbf{t}) = C(f(\mathbf{r}, \mathbf{v}, \mathbf{t})) \tag{2.1}$$

It calculates the probability density function f, which is a function of space r, time t and the velocity v. Streaming forms the backbone of this project, and can be discretized as in equation 2.2.

$$f_i(\mathbf{r} + \mathbf{c}_i\Delta t, t + \Delta t) = f_i(\mathbf{r}, t) + C(\mathbf{r}, t) \tag{2.2}$$

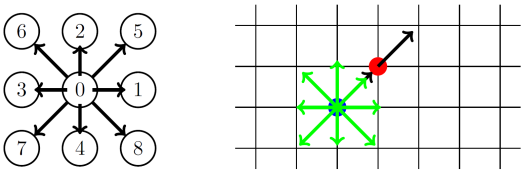
Streaming alone represents the movement of particles in vacuum without considering how these particles interact with each other; or in other words without collision [6]. The direction of streaming is governed by velocity channels, which are collected in a single array c. The velocity sets c_i and the discretization Δx and Δt are chosen so that $c_i\Delta t$ points from the current node to the neighbouring ones $r + c_i\Delta t$ [6]. Each column in this matrix represents one of the nine directions for streaming. We use:

$$c = \begin{pmatrix} 0 & 0 & -1 & 0 & 1 & -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \end{pmatrix} \tag{2.3}$$

Figure 2.1 provides a visualization for the streaming step.

2.2 Collision

The right hand side of equation 2.1 is the collision term. The collision term defines how each particle in the lattice will interact with each other. It is usually a two-part scattering



(a) (b)

Figure 2.1: Discretization of the BTE. (a) Discretization of velocity space into nine directions. The numbers uniquely identify the direction. (b) Regular two-dimensional lattice used for the spatial discretization [6]

.

integral. Since it is complex, we approximate this process through a relaxation constant. The collision term from equation 2.2 can be expanded into 2.4:

$$C[f(\mathbf{r}, \mathbf{v}, \mathbf{t})] = -\omega[f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)] \tag{2.4}$$

We relax the probability density function f towards the equilibrium function f^{eq} in collision, this is controlled with the relaxation parameter $\omega = \Delta t/\tau$. We define the equilibrium density function as:

$$f_i^{eq}(\mathbf{x}, t) = w_i \rho(\mathbf{x}, t)[1 + 3c_i \cdot \mathbf{u}(\mathbf{x}, t) + \frac{9}{2}(c_i \cdot \mathbf{u}(\mathbf{x}, t))^2 - \frac{3}{2}||\mathbf{u}(\mathbf{x}, t)||^2] \tag{2.5}$$

Here, w_i is defined for the D2Q9 lattice as $w = (\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36})$

2.3 Boundary Conditions

Boundaries, while forming a small subset of interactions in the implementation, play an important role for the dynamics of the system. There are broadly two arrangements when working with boundaries in the LBM. They are:

- Dry node: Boundaries occur at the link between the nodes
- Wet node: Boundaries occur at the lattice points themselves

In this implementation, we focus on the dry node arrangement. This is because it is easier to be implemented and they retain a second order accuracy as long as the physical wall is placed exactly halfway between the nodes [1]. We encounter four kinds of boundary conditions. They are:

2.3.1 Periodic boundary

The periodic boundary condition asserts that particles flowing out of the domain on one side shall re-enter the domain on the other side [7]. It is useful in situations where we need to predict the flow over a finite section of a flow field. It can be written as:

$$f_i(\mathbf{x}_1, t) = f_i(\mathbf{x}_n, t) \tag{2.6}$$

2.3.2 Periodic boundary with pressure gradient

A modification of the aforementioned periodic boundary conditions involves a prescribed pressure variation Δp between the inlet and outlet. The pressure is related to the fluid density as $p = c_s^2 \rho$, where $c = 1/3$ in lattice units [6].

To implement this condition, we need to add virtual layers outside the boundaries, i.e, before the first and after the last nodes. The input and output pressures (p_{in} and p_{out}) need to be assigned. The equations for the equilibrium are:

$$f^{eq}(x_0, y, t) = f_i^{eq}(\rho_{in}, u_N) \tag{2.7}$$
$$f^{eq}(x_{N+1}, y, t) = f_i^{eq}(\rho_{out}, u_1) \tag{2.8}$$

where n is the length of the domain. Thus, our extra virtual nodes get assigned to index 0 and $n + 1$. We also have a component called the non-equilibrium distribution, which is calculated after collision. It is defined as $f^{neq} = f_i^* - f_i^{eq}$, where$f_i^*$ represents the population before streaming [7]. Thus, we define the non-equilibrium equations as:

$$f^{neq}(x_0, y, t) = f_i^{neq}(x_N, y, t) \tag{2.9}$$
$$f^{neq}(x_{N+1}, y, t) = f_i^{neq}(x_1, y, t) \tag{2.10}$$

Using these definitions, we can now write the equations for the inlet and outlet boundary condition. The inlet and outlet boundary conditions are defined as:

$$f_i^*(x_0, y, t) = f_i^{eq}(\rho_{in}, \mathbf{u}_N) + (f_i^*(x_N, y, t) - f_i^{eq}(x_N, y, t) \tag{2.11}$$
$$f_i^*(x_{N+1}, y, t) = f_i^{eq}(\rho_{out}, \mathbf{u}_1) + (f_i^*(x_1, y, t) - f_i^{eq}(x_1, y, t) \tag{2.12}$$

These equations assert that considering flow in the x-direction, the population at node 0 is identical to the population at node N. Similarly, the population at node 1 is identical to node $N + 1$.

2.3.3 Rigid wall

The rigid wall boundary condition, as the name suggests, introduces a bounce-back scheme to the system and ceases the property of periodicity in practice (although in theory, periodicity still occurs). In this condition, we need to consider both the pre-streaming and post-streaming populations. The underlying principle is simple. We set the value of the post-streaming population at a wall to the value of the pre-streaming population at the wall but of the diagrammatically opposite channels. As an example: for each streaming step, we set the values for channels 4, 7 and 8 of the population at top wall at time-step $t + \Delta t$ to the values of channels 2, 6 and 5 of the same population at time-step t. This sequence of events can be generalized as:

$$f_{\bar{i}}(x_b, t + \Delta t) = f_i^*(x_b, t) \tag{2.13}$$

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

2.3.4 Moving wall

The moving wall boundary condition is similar to the rigid wall in that it also uses the same principles of setting the post-streaming population values to that of the pre-streaming population at opposite channels, but it introduces an additional term which is dependent on the wall velocity. This is input by the user. It dictates the velocity of the flow parallel to the wall. The equation is as follows:

$$f_{\bar{i}}(x_b, t + \Delta t) = f_i^*(x_b, t) - 2w_i\rho_w \frac{\mathbf{c}_i u_w}{c_s^2} \tag{2.14}$$

The density in this equation is the wall density.

3

Implementation

In this implementation, the lattice grid is defined with the dimensions `grid_x` and `grid_y` which denote the number of rows and columns respectively. The density matrix `rho_nm` matches the grid shape, and is defined below.

$$\rho(\mathbf{x}_j, t) = \sum_i f_i(\mathbf{x}_j, t) \tag{3.1}$$

The velocity, defined in equation 3.2, is initialized as an array of shape `(2, grid_x, grid_y)`.

$$\mathbf{u}(\mathbf{x}_j, t) = \frac{1}{\rho(\mathbf{x}_j, t)} \sum_i \mathbf{c}_i f_i(\mathbf{x}_j, t) \tag{3.2}$$

Each of these components can be derived from f, which we represent as `f_inm`, of shape `(9, grid_x, grid_y)`. It is the matrix for the probability density function. Shown in listing 3.1, for each time-step, we have a set sequence of operations containing all the components defined in Chapter 2. We repeat this sequence for as many time-steps defined.

Listing 3.1: Time step

```
for i in range(time_steps):
    f_inm, u_amm = collision(f_inm, c_ai, w_i, omega)
    f_inm = Communicate(f_inm, cartcomm, sd)
    f_inm= boundary(f_inm, rho_nm, c_ai, w_i, boundary)
```

3.1 Streaming

We have a streaming function which is called inside the boundary function. Streaming is easily performed using the built-in `np.roll` function. The streaming direction is controlled by `c_ai` as defined in 2.3.

Listing 3.2: Streaming function

```
def streaming_operator(f_inm, c_ai):
    for i in range(9):
        f_inm[i] = np.roll(f_inm[i], shift=c_ai.T[i], axis=(0, 1))
    return f_inm
```

3.2 Collision

For each time-step, we pass our population density function (pdf) and velocity matrix into the collision function. The collision function calls the equilibrium function within it which calculates the equilibrium distribution:

Listing 3.3: Collision function

```
def collision(f_inm, c_ai, w_i, omega):
    rho_nm = np.einsum('ijk->jk', f_inm)
    u_anm  = np.einsum('ai,inm->anm', c_ai, f_inm) /  rho_nm
    feq_ixy = feq(rho_nm, u_anm, c_ai, w_i)
    return f_inm + (omega * (feq_ixy - f_inm)), u_anm
```

The equilibrium function is `feq` and is defined as:

Listing 3.4: Equilibrium function

```
def feq(rho_nm, u_anm, c_ai, w_i):
    cu_nm = np.einsum('ai,anm->inm', c_ai, u_anm)
    sqcu_nm = cu_nm**2
    usq_nm = np.einsum('anm,anm->nm', u_anm, u_anm)
    wrho_inm = np.einsum('i,nm->inm', w_i, rho_nm)
    feq_inm = wrho_inm * (1 + 3 * cu_nm + 4.5 * /
    sqcu_nm - 1.5 * usq_nm)
    return feq_inm
```

Note that to perform the summations along axes in the calculations of the density matrix, or in the future for any similar operations, we use an Einstein summation instead, which is denoted by the keyword `np.einsum`. Thus, our order of operations proceeds as: calculating the equilibrium distribution, accounting for collision, communicating between processes, streaming, and finally applying boundary conditions. The communication function is explained in section 3.4.

3.3 Boundary Handling

While periodicity is implicitly applied with `np.roll`, for the rigid and moving wall we need to use the pre-streaming population to apply the conditions. We do this by storing it in a

dummy variable. We then perform a streaming step and apply the boundary conditions. To determine which boundaries use the rigid or moving walls, we define arrays which store the channel numbers, and depending on which boundary it is, we use the corresponding arrays. There are four arrays for four directions and each array stores three integers.

Listing 3.5: Rigid wall implementation

```
if boundary[0] == True:
 if i in index_w:
   f_inm[index_e[np.where(i==index_w)[0][0]],:,0] = /
   f_inm_dummy[i, :, 0]
```

As in listing 3.5, we use an array `index_w` (shortened from `index_west` in the actual implementation) which represents the channels facing left. This depicts the application of a rigid wall on the left boundary. `f_inm_dummy` represents the population before streaming (`index_e stands for index_east`. The algorithm for the moving wall condition is identical with an added term, as shown below:

Listing 3.6: Moving wall implementation

```
if boundary[3] == True:
 if i in index_north:
   f_inm[index_s[np.where(i == index_n)[0][0]], 0, :]=/
   f_inm_dummy[i, 0, :] - 2 * w_i[i] * rho_average * /
   ((c_ai[:, i] @ u_w).T / (c_s ** 2))
```

Equation 2.14 mentions a wall density ρ_w. In this implementation, we set it to the average density of the population. `u_w` stands for the wall velocity.

Periodic boundaries with pressure variation needs a different approach. The function needs an additional input in the inlet and outlet densities. Since we do not input the pressure variation directly, we calculate it through the densities as $p = c_s^2\rho$. We also define extra nodes at the inlet and outlet as explained in section 2.3.2. The implementation in listing 3.7 is exactly as outlined in equations 2.11, with intermediate terms `f_int_beg` and `f_int_end` added in for improved readability. Note that `feq_limited` is simply a modified version of the equilibrium function from listing 3.4.

Listing 3.7: Periodic boundary with pressure variation

```
'''calculate inlet pdf'''
f_eq_in = feq_limited(rho_in, u_out, c_ai, w_i)
f_int_beg = f_inm[:, :, -2] - feq(rho_nm[:, -2], u_out, c_ai, w_i)
f_inm[:, :, 0] = f_eq_in + f_int_beg

'''calculating outlet pdf'''
f_eq_out = feq(rho_out, u_in, c_ai, w_i)
f_int_end = f_inm[:, :, 1] - feq(rho_nm[:, 1], u_in, c_ai, w_i)
```

```
f_inm[:, :, -1] = f_eq_out + f_int_end
```

3.4 Parallellization

The LBM runs well in parallel. As mentioned in section 1.1, we use the `ipyparallel` and `mpi4py` packages. To enable it to do so, we decompose it in the spatial domain using the message passing interface MPI. This is done by using the cartesian communicator from MPI, initialized as `cartcomm` in 3.8. For our system, we use 9 processes locally. For a 300x300 domain, it yields a 3x3 subdomain grid. After domain decomposition, the different processes need to communicate with each other at every step, as opposed to working serially. The `Communicate` function from listing 3.1 defines this operation. We have only included a snippet for a communication step leftwards and downwards. The first step is to pad each subdomain with "ghost cells". These cells are outside the physical domain and are used solely for communication. At each communication step, the population at the boundary of a subdomain gets "communicated" to the ghost cell of its adjacent subdomain. This is done via a `recvdbuf` array which copies the boundary node of the physical domain into it and through a `sendrcv` call, it gets sent to its destination, which is the adjacent ghost cell. Note that in listings 3.5 and 3.6, we check if `boundary[i]` is `True`. This is a simple check to see if the boundary is located at the outermost boundary of the domain, as opposed to the inner boundaries of one of the subdomain. The distinction is that for inner boundaries, we communicate without performing boundary handling. At the outermost boundaries, we need to apply boundary conditions. After this step, streaming must be performed so that the information from the ghost cell reaches the physical domain of the new process. The communication operation for a 2x3 domain decomposition is illustrated in 3.1, taken from [6]. The `Communication` function is defined as:

Listing 3.8: Communication function

```
cartcomm = comm.Create_cart(dims=[sectsX, sectsY],
           periods=[False, False], reorder=False)

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

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

  '''Send down and receive from up'''
```

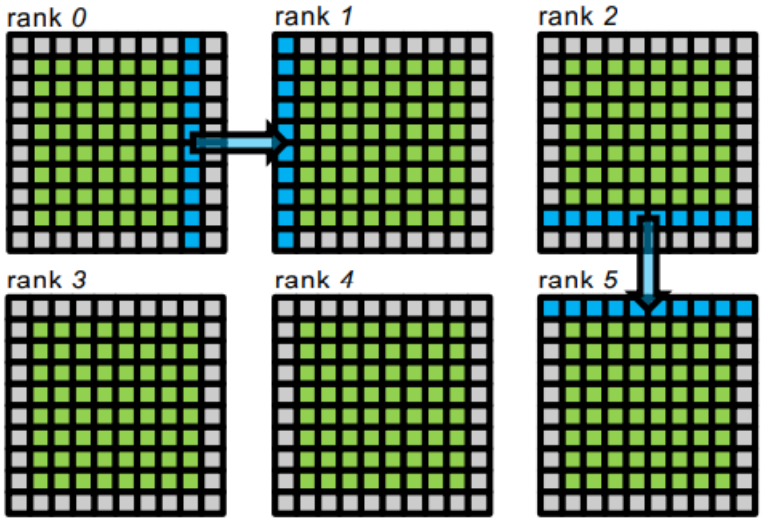


Figure 3.1: Communication between ranks. The green cells are part of the physical domain, and the greyed out cells are the ghost layers. A rightward and downward communication step is illustrated. The boundaries of rank 0 and rank 2 get communicated to the adjacent ghost cells

```
sendbuf = np.ascontiguousarray(c[:, -2, :])
recvbuf = c[:, 0, :].copy()
cartcomm.Sendrecv(sendbuf=sendbuf, dest=dD, /
recvbuf=recvbuf, source=sD)
c[:, 0, :] = recvbuf
```

As in listing 3.8, we similarly use `sendrcv` calls for communication upwards and rightwards. Diagonal communication is implicitly performed with this operation. It must be emphasized that streaming should only be performed after the communication step.

4

Experiments and Results

In this section, a series of experiments is conducted on our system, and their results are analyzed. Each experiment is self-contained within its own Python file and can be run independently. The one exception is the parallelized version of the sliding lid experiment which needs certain considerations as is outlined in section 4.5

4.1 Shear Wave Decay

The shear wave decay experiment demonstrates the effect of a sinusoidal shear wave perturbation on the evolution of the density and velocity until the steady state. This convergence to the steady state is aided by the viscosity, which we calculate through ω. We conduct this experiment in two parts, the first for observation of density, and the second for velocity. In order to obtain an analytical description of this phenomenon, we consider the Navier-Stokes equations given by:

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

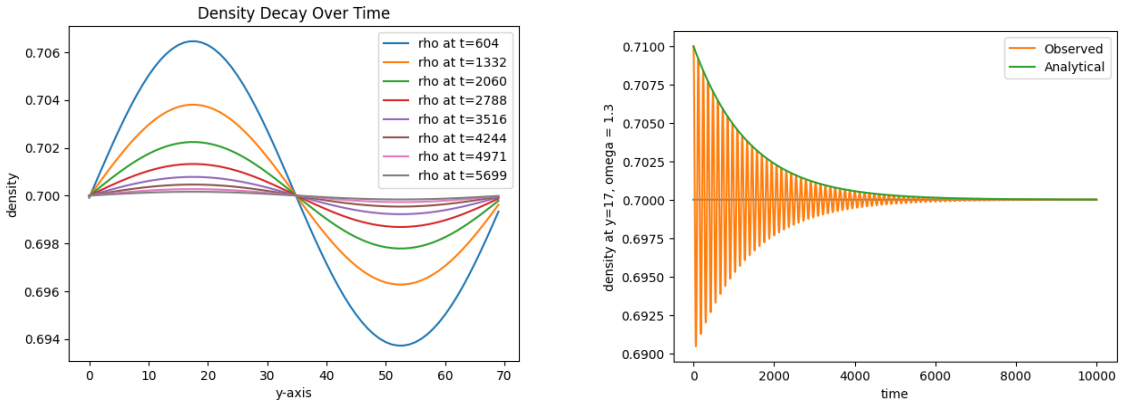
However, since there is no analytical solution, we make two assumptions as follows:

1. Because the perturbation is small, we can conclude that the velocity gradient is small enough to ignore the non-linear term $(u.\nabla)u$
2. The pressure gradient Δp can also be ignored on account of it being too small

These assumptions can now give us the analytical solution, which we use in our experiments for validation:

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

This equation describes the exponential decay of the density or velocity after an initial perturbation as explained in the coming subsections. The amplitude a_0 is set to either the initial density in case of a density decay or zero in case of velocity decay. ν is the viscosity, dependent on ω through the analytical formula $c_s^2(\frac{1}{\omega} - \frac{1}{2})$.



(a) Density decay over time

(b) Density numerical decay vs. analytical solution

Figure 4.1: Shear wave decay (density)

For this experiment, we consider a grid of size 30x70 for the density case and 70x30 for the velocity case ($X * Y$ form, where X is the number of rows and Y is the number of columns). ω=1.3, ϵ=0.01, ρ_0=0.7. The plots are constructed by isolating the peak of each wave. These peaks are compared to the analytical solution. In both cases, it is evident that the results obtained are accurate as they are perfectly validated by the analytical values.

4.1.1 Density

The density initialization at $t = 0$ is dictated by the formula:

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

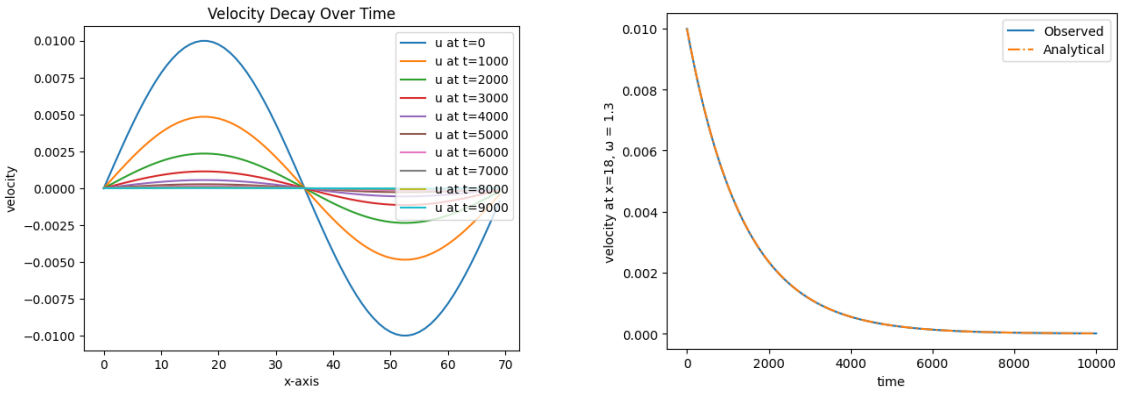
Where, $0 < \rho_0 < 1$ is the initial density and L_x is the length of the domain in the x direction. As can be surmised from the equation, a perturbation in the form of a sinusoidal wave is introduced and we observe the evolution of the density profile over time. ϵ represents the amplitude of the wave. In figure 4.1(a), we see the results of the density decay over 10000 time-steps, and in 4.1(b) we see how the decay compares to the analytical solution.

4.1.2 Velocity

In the case of velocity shear wave decay, we now initialize our velocity $\mathbf{u_0}$ at time $t = 0$ to:

$$u(\mathbf{r}, 0) = \epsilon \sin \frac{2\pi x}{L_y} \tag{4.4}$$

We use the same parameters (with the exception of grid size as mentioned previously) and constants as the density version, and similar to the density, we can observe our velocity



(a) Velocity decay over time

(b) Velocity numerical decay vs. analytical solution

Figure 4.2: Shear wave decay (velocity)

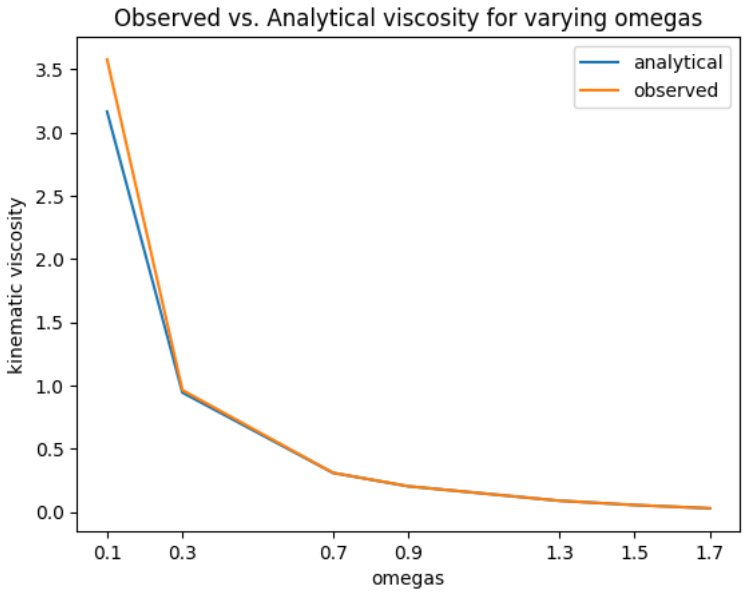


Figure 4.3: We compare the analytical solution for viscosity vs the observed kinematic viscosity for the mentioned parameters and for ω=0.1, 0.3, 0.7, 0.9, 1.3, 1.5, 1.7.

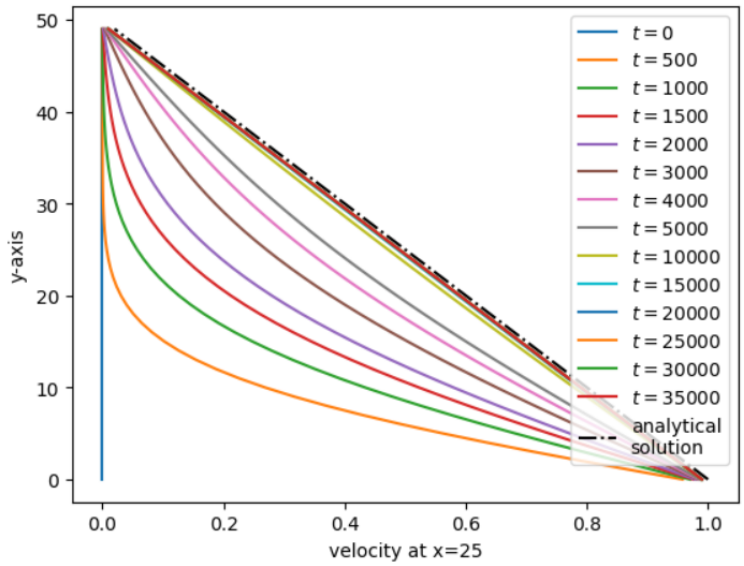


Figure 4.4: Velocity profile for Couette flow at staggered time-step intervals. Lattice grid size is 50x50, ω=1.3 and measured over 40000 time-steps. The top wall is moving and the bottom is rigid.

slowly decay to the steady state in 4.2(a), as can be verified by the analytical solution in 4.2(b).

In figure 4.3, we can see that the measured kinematic viscosity matches that of the analytical solution for viscosity.

4.2 Couette Flow

Couette flow simulates a fluid with a prescribed viscosity ν as it flows between two walls, one of which is rigid and the other is moving. The remaining two walls have periodic boundary conditions applied to them. For this experiment, the bottom wall is considered rigid, the top is considered moving, and the others, periodic.

The moving wall is controlled by a wall velocity U_w, and because of the fluid's viscosity, it triggers a flow which eventually comes to a steady state. This is defined in the analytical solution below:

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

We verify our results by plotting it against the analytical solution in figure 4.4. After a certain time step, it should converge to it. For this implementation, we consider $U_w = 1.0$ and $\omega = 1.3$.

4.3 Poiseuille Flow

Poiseuille flow is a slight modification of Couette flow. It uses periodic boundary with pressure variations on the left and right boundaries. It is this pressure gradient that induces a flow like in a pipe. There exists an analytical solution for the velocity field for this flow in a pipe, called the Hagen-Poiseuille flow. We first simplify the Navier-Stokes equations and consider only the streamwise component x aligned to with the pipe axis. We then integrate it in the y-direction twice and use the boundary conditions $u(0) = 0$ and $u(h) = 0$ to obtain the velocity profile.

$$u(y) = \frac{1}{2\mu}\frac{dp}{dx}y(h-y) \tag{4.6}$$

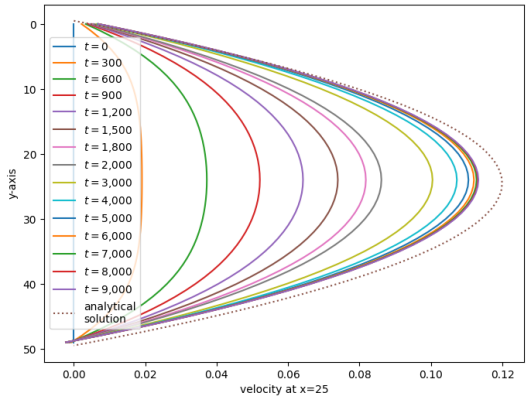
In equation 4.3, $\mu = \rho\nu$ and ρ is the dynamic viscosity. For our experiment, we consider a 50x50 grid. We pad the grid along the x-axis as defined in section 2.3.2. We do not input the pressure directly to get the pressure difference, but calculate it from the density according to the formula $p = c_s^2\rho$, where $c = 1/3$. For this purpose, we input the inlet and outlet densities. We initialize the density as $\rho_0(0)$=1 and velocity as $\mathbf{u}(0) = 0$ at time t=0. The inlet and outlet densities are set to 1.005 and 0.995 respectively. This setup is run over 10000 time-steps, and a plot of the evolution of the velocity profile is visualized in figure 4.5(a), with the flow field visualized in 4.5(b). The area of the velocity profile and density profile across the center are plotted in figure 4.6.

4.4 Sliding Lid

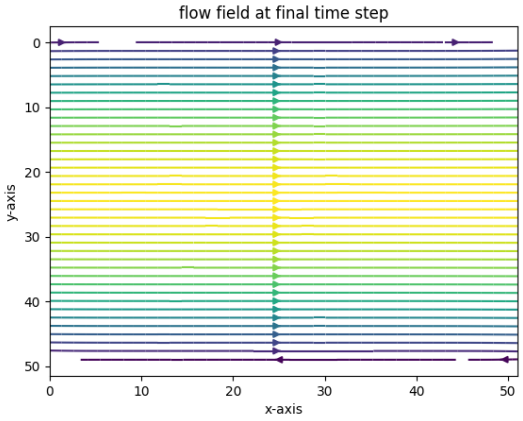
The sliding lid system is a test frequently used in fluid dynamic simulations [6]. It is described as a closed box with a sliding lid. The bottom, right and left walls use the rigid wall boundary conditions and the top wall is a moving wall with a wall velocity U_w to the east. A key component in this experiment is a dimensionless constant called Reynold's number (Re). The law of similarity in fluid dynamics states: two incompressible flow systems are dynamically similar if they have the same Reynolds number and geometry [8]. Reynold's number is commonly defined as:

$$Re = LU/\nu \tag{4.7}$$

This serves as the equation for our calculations where L is the length of the system, U its velocity, and ν its kinematic viscosity. The aim is to visualize the particles in the system after the lid is slid over it with a given velocity until the steady state. This experiment yields a distinct flow pattern with smaller circular currents at the bottom corners. We initialize $\rho_0(0)$=1 and $\mathbf{u}(0) = 0$. We benchmark different configurations, mainly with Re=1000, with varying values for ν and ω. However, when testing serially, or even locally in parallel, we try out smaller lattice sizes owing to computational limits, as these runs can get exponentially computationally expensive. In figure 4.7, some results for the serial implementation of this experiment are shown for lattice sizes 75x75, 150x150 and 300x300, at various time-steps.

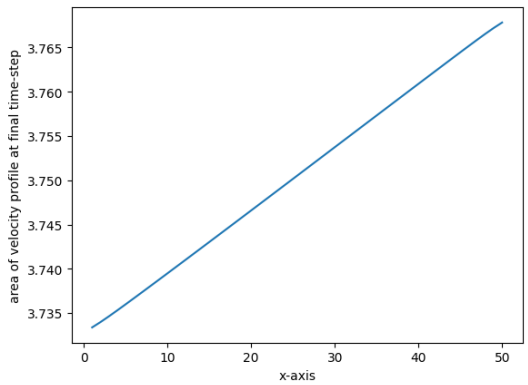


(a) Velocity profile evolution over 10000 time-steps

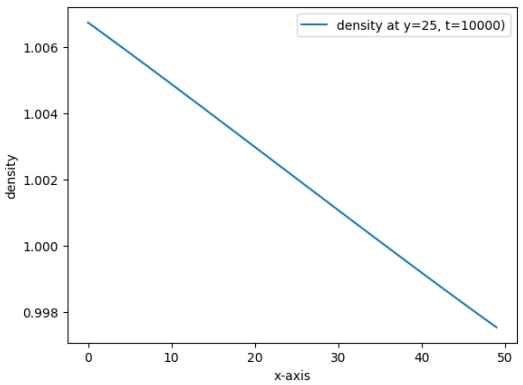


(b) Velocity flow field at final time step

Figure 4.5: Poisouille flow on a 50x50 grid with ω=1.0. Inlet and outlet densities are 1.005 and 0.995 respectively. The velocity profile almost converges with the analytical solution shown by the dotted curve. Just like in the Couette flow experiment, we use staggered time steps to plot the velocity profile. For further validation, we show the generated flow field with streamlines at the final time step in figure 4.6(a). The velocity is the maximum at the center of the plot where it is yellow, as that represents the maximum velocity equal to U_w.

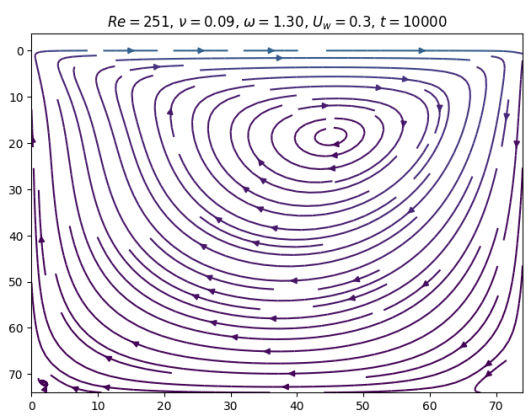


(a) Area of velocity profile from left to right at last time step

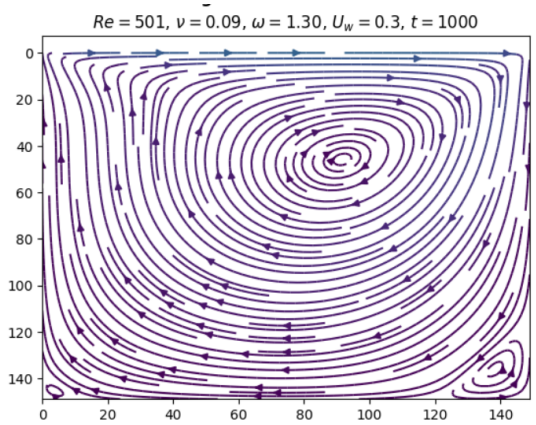


(b) Density profile across the center at the last time-step

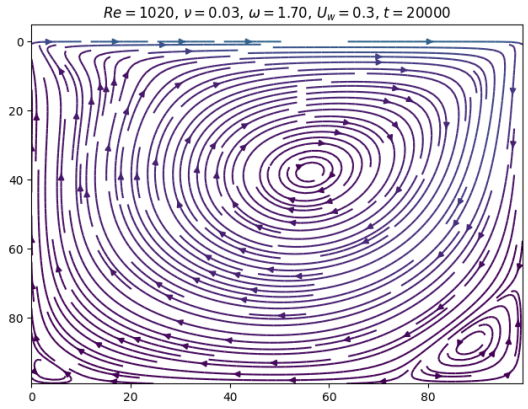
Figure 4.6: We plot the velocity along the x-axis, and we can observe the area of the velocity profile. The precise percentage increase in the area is 0.922 (b) depicts the density along the centerline at $\rho(x, 25)$. It linearly decreases along the x-axis from the inlet to the outlet.



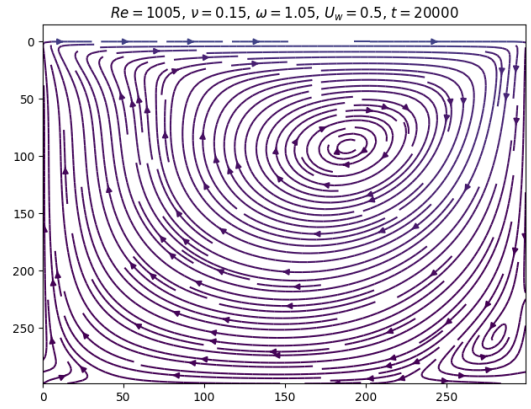
(a) Re≈250 for a 75x75 grid



(b) Re≈500 for a 150x150 grid



(c) Re≈1000 for a 100x100 grid at 20000 time-steps



(d) Re≈1000 for a 300x300 grid at 20000 time-steps

Figure 4.7: In the serial implementation of the sliding lid, we can see the distinct swirls forming early on. At 1000 time-steps we can observe, the swirls at the corners beginning to form. At Re≈250, it is not as distinct as in cases with Reynold's numbers close to 1000. The runtime in this setup was extremely poor, sometimes taking over an hour to run, especially with an increasing number of time-steps.

To parallelize this experiment, we use 9 processes locally with `ipyparallel` as shown in figure 4.8(a) and 4.8(b). On the BWUniCluster, we are not inhibited by CPU power, and can freely test 300x300 and 500x500 with 100000 time-steps (shown in 4.8(c) and (d)). We test each of those grids with an increasing number of processes, from 4 up to 1024. We then calculate the million lattice updates per second (MLUPS) that it took for each of those cases. The equation is given by:

$$MLUPS = \frac{L_x \cdot L_y \cdot steps}{t} \tag{4.8}$$

Figure 4.9 shows how the system scales with the number of processes on both the 300x300 and 500x500 grids. With processes under 50, the increase in MLUPS is exponential. There is a decrease on both the grids around the 75-100-process mark, however the graph resumes its original trajectory, showing further exponential increase until 250 processes. Following that, it plateaus and starts decreasing after the 400 mark, and drops further around the plotted 784 and 900 processes. Thus, we see that the MLUPS peaks at the 400-process mark.

Here, L_x and L_y are the dimensions of the grid, t is the total time taken for the calculation, and *steps* is the number of time-steps.

4.5 Notes on Reproducibility

4.5.1 Local, Serial

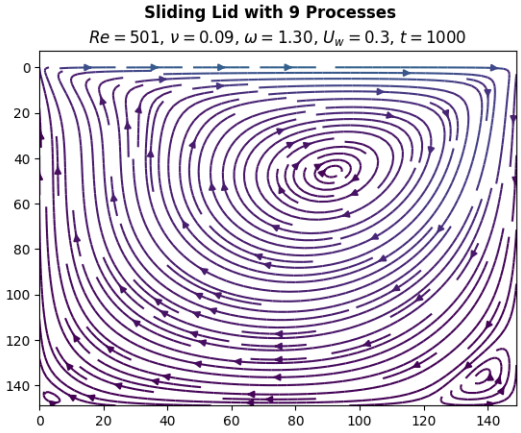
As mentioned in the beginning of the previous chapter, every experiment is self-contained and can be run independently. All the Python files and job scripts used are provided. Every experiment has explicitly declared parameters like the grid size as explained in chapter 3. All constants have been declared as self-explanatory variables like `nu, omega, epsilon`, etc. which can be easily located and modified to carry out further tests. Thus, all results can be verified.

4.5.2 Local, Parallel

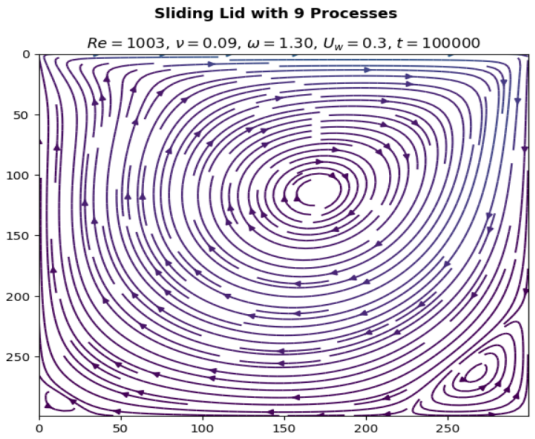
On the parallelized version of the sliding lid experiment, there is one further step to visualize the result. We use the function `save_mpiio()` to save the results obtained to `.npy` files. We store the x and y components of the velocity field in these files. We use `plot.py` to load both of the `.npy` files and make the necessary streamplot. For the local parallelized version, they are located in the home directory. Loading and plotting these files in `plot.py` results in the plots seen in 4.8(a).

4.5.3 BwUniCluster

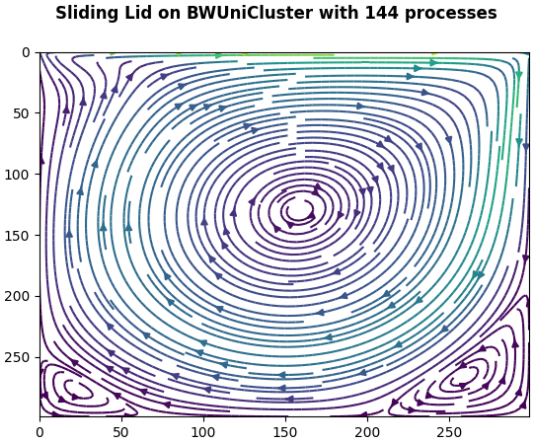
To run the simulations on the cluster, we use the `job_100.sh` to load modules and to ask for the processes we need. This runs the Python file for the experiment named



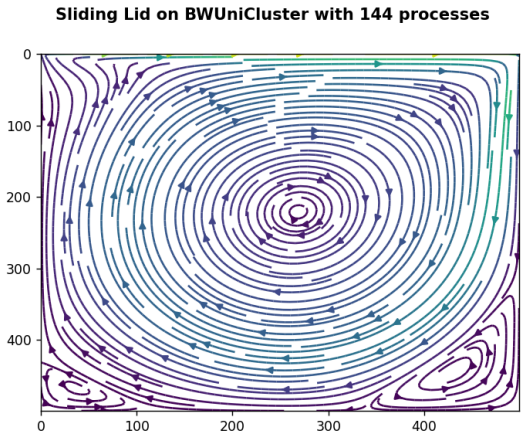
(a) Re≈500 for a 150x150 grid at 1000 time-steps



(b) Re≈1000 for a 300x300 grid with 144 processes



(c) Re≈1000 for a 300x300 grid at 100000 time-steps with ω=1.7, ν=0.5 and U_w=0.03 with 144 processes on BWUniCluster



(d) Re≈1000 for a 500x500 grid with ω=1.7, ν=0.5 and U_w=0.03 at 100000 time-steps with 144 processes on BWUniCluster

Figure 4.8: Parallelizing the sliding lid experiment is necessary, as the problem itself is parallel in nature. In these cases, we use 9 processes to plot the observe the streamplots of Re≈500 and Re≈1000. The compilation times were much better but still not optimal, with the configuration at 100000 time-steps taking a few hours to run. The BWUniCluster sped this up massively, never taking more than 30 minutes at a time to run, which worked perfectly with the channels that were requested. The swirls at the bottom corners are prominent.

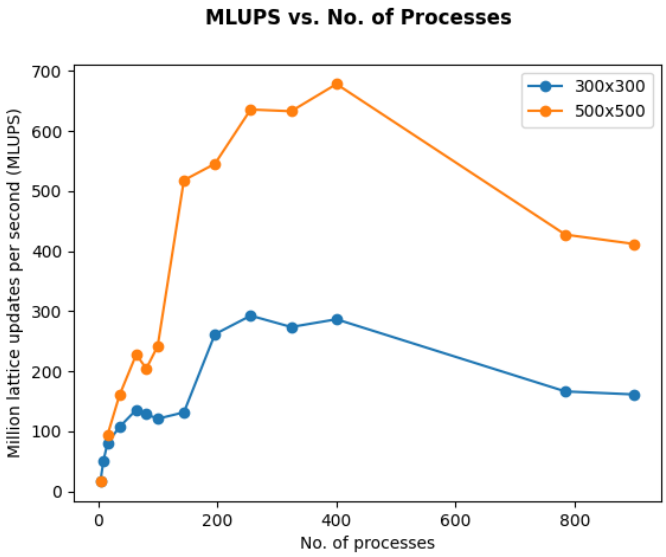


Figure 4.9: The MLUPS vs. number of processes plot for 300x300 and 500x500 grids. We set Re≈1000, ω=1.7, U_w=0.1 and ν≈0.03 for 300x300 and Re≈1000, ω=1.7, U_w≈0.2 and ν=0.1 for 500x500. Both simulations are run for 100000 time-steps.

`sliding_lid_parallel.py`. On running this, it generates `.out` and `.npy` files, which we named according to the grid size and number of processes we used and save them to the folder /`MLUPS` which output the values of the MLUPS obtained as well as store runtime and user information. Not every run was documented in this form.

Within the /`BWUniCluster` folder, there are two sets of `ux_xxx.npy` and `uy_xxx.npy`, ending with either 300 or 500. Loading and plotting either set in `plot.py` yields results on a 300x300 or 500x500 grid size with 144 processes as seen in 4.8(c) and 4.8(d),

The results from the `.out` files are manually copied to the `plot_MLUPS.py`, so it can be run locally to display the graph in 4.9. More information can be found in `ReadMe.txt`.

5

Conclusion

This report outlined the Lattice Boltzmann Method and the Boltzmann transport equation. Methods like streaming, collision, boundary handling and parallelization were introduced and discussed. Results from the applications of these methods were discussed in the shear wave decay experiment, Couette flow, Poiseuille flow, and the sliding lid experiment. Some of these were compared to the analytical solutions, and all of them were found to satisfy these comparisons.

The next part dealt with parallelization using the BWUniCluster 2.0. Run-time of the experiments was cut short exponentially, especially over large time-steps. The MLUPS plot to test for scalability of the number of processes proved conclusive. Increasing the number of processes under the 100 mark showed a sharp rise in the MLUPS before plateauing after the 400 mark and then steadily decreasing until 900 processes.

Bibliography

[1] Dieter A Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2004.

[2] Yuanxun Bill Bao and Justin Meskas. Lattice boltzmann method for fluid simulations. *Department of Mathematics, Courant Institute of Mathematical Sciences, New York University*, 44, 2011.

[3] 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.

[4] Shiyi Chen and Gary D. Doolen. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

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

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

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

[8] Ira M Cohen and Pijush K Kundu. *Fluid mechanics*. Elsevier, 2004.