High Performance Computing: Fluid Mechanics with Python

– Final Report –

Peter Würth
5306261
peter.wuerth@students.uni-freiburg.de

August 13, 2022

Contents

1 Introduction **1**

2 Methods **2**

2.1 Boltzmann Transport Equation 2

2.2 Lattice Boltzmann Method (2D) 3

2.3 Boundary Handling 4

2.3.1 Rigid Wall 5

2.3.2 Moving Wall 5

2.3.3 Periodic Boundary Conditions with Pressure Gradient . . 6

3 Implementation **7**

3.1 Lattice Representation 7

3.2 Simulation Step 7

3.2.1 Streaming 8

3.2.2 Collision 8

3.2.3 Boundary Conditions 9

3.3 Parallelization 10

4 Experiments **13**

4.1 Shear Wave Decay 13

4.2 Couette Flow 17

4.3 Poiseuille Flow 17

4.4 Lid-Driven Cavity 19

4.5 Scaling 20

5 Conclusions **23**

1 Introduction

The Lattice Boltzmann Method (LBM) is a numerical scheme based on a discretization of the Boltzmann Transport Equation (BTE) used to simulate fluid dynamics, introduced by McNamara and Zanetti [1].

This report discusses an implementation of the LBM and analyses its scaling properties in the context of the course *High-Performance Computing: Fluid Mechanics with Python* at the University of Freiburg. The implementation is done completely in Python using the NumPy[1] [2] library for efficient numerical calculations and the Message Passing Interface (MPI)[2] [3] with mpi4py[3] [4] as bindings for the parallelization. For simplicity, the problem is restricted to the 2D case.

The code associated with this report as well as introduction on how to run it can be found on GitHub[4].

[1] NumPy version 1.23
[2] MPI version 4.0
[3] mpi4py version 3.1
[4] https://github.com/petwu/hpc-with-python

2 Methods

This chapter describes the Lattice Boltzmann Method (LBM) and its underlying concept, the Boltzmann Transport Equation (BTE).

2.1 Boltzmann Transport Equation

The BTE describes the statistical behavior of particles in system involving density or temperature gradients such as e.g. fluids. It formulates the probability of finding a particle with velocity v at position r over time t using the probability density function (PDF) $f(r, v, t)$. Both r and v are functions of time t themselves. The function f is related to macroscopic values like the density ρ or the fluid velocity u through its moments, i.e. integrals over f weighted with some function of v [5]:

$$\rho(r,t) = \int f(r,v,t) d^3v \tag{1a}$$

$$u(r,t) = \frac{1}{\rho(r,t)} \int v f(r,v,t) d^3v \tag{1b}$$

The BTE representing time-rate of change of f is given by the following equation [6]:

$$\frac{df}{dt} = \frac{\partial f(r,v,t)}{\partial t} + \underbrace{v\frac{\partial f(r,v,t)}{\partial r} + a\frac{\partial f(r,v,t)}{\partial v}}_{\text{streaming term}} = \underbrace{\left(\frac{\partial f(r,v,t)}{\partial t}\right)_{coll}}_{\text{collision term}} \tag{2}$$

The l.h.s. of the equation is often called the streaming term and describes the movement of particles over time, while r.h.s. is called the collision term which accounts for all interactions between particles. Boltzmann's original collision operator is a complex double integral over velocity space that considers all the possible outcomes of two-particle collisions. It is therefore common [5], to use a relaxation time approximation for the collision term, which assumes that f locally relaxes towards an equilibrium distribution f^{eq} with a characteristic relaxation time constant τ as suggested by Bhatnagar, Gross, and Krook (BGK) [7]:

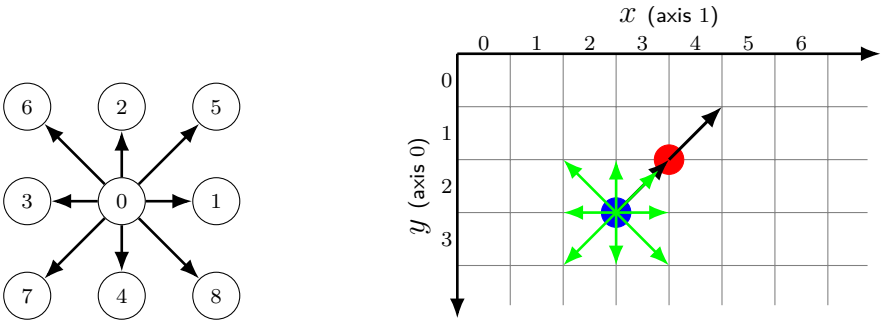
$$\frac{d}{dt} f(r,v,t) = -\frac{f(r,v,t) - f^{eq}(v,\rho,u,T)}{\tau} \tag{3}$$

2.2 Lattice Boltzmann Method (2D)

The LBM is a discretization of the BTE in space, velocity and time in order to solve the problem numerically.

Discretization in the spatial domain is straightforward by dividing the space into an equidistant 2D grid, the lattice, as depicted in Figure 1b. For the velocity space and time, the discretization is chosen such that the distance between interpolation points in velocity space c_i multiplied by the time step Δt equals the distance between points on the spatial lattice Δx_i, i.e. $c_i \Delta t = \Delta x_i$, i.e. the velocity grid and the spatial grid should be coincident. For this purpose, a velocity set with nine directions, or channels, as illustrated in Figure 1a is chosen. In combination with the velocity channels in Equation (4) and unity dimensions $\Delta x = \Delta y = \Delta t$ the above condition is fulfilled. This particular discretization is commonly referred to as D2Q9, since it discretizes a 2-dimensional space with 9 velocity channels c_i [5].

$$c = \begin{pmatrix} 0 & 0 & -1 & 0 & 1 & -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \end{pmatrix}^T \tag{4}$$



(a) Velocity discretization: D2Q9 **(b)** Spatial discretization: regular 2D lattice

Figure 1: Discretization of the BTE

The probability distribution $f(r, v, t)$ can now be expressed as a set of discrete occupation numbers $f_i(x_j, t)$ for each of the nine velocity channel c_i per lattice point x_j. The moments of f given in Equations (1a) and (1b) can then be

discretely expressed as follows [5]:

$$\rho(x_j, t) = \sum_i f_i(x_j, t) \tag{5a}$$

$$u(x_j, t) = \frac{1}{\rho(x_j, i)} \sum_i c_i f_i(x_j, t) \tag{5b}$$

The discrete BGK-variant of the Boltzmann equation [7] is then given by

$$\underbrace{f_i(x_j + c_i \Delta t,\ t + \Delta t) - f_i(x_j, t)}_{streaming} \ = \ \underbrace{-\omega \left[f_i(x_j, t) - f_i^{eq}(x_j, t) \right]}_{collision} \tag{6}$$

where $\omega = \frac{1}{\tau}$ is the relaxation rate. The equilibrium distribution function is stated in Equation (7a) [6] with the weights w_i for the corresponding velocity channels c_i.

$$f_i^{eq}(x_j, t) = w_i \rho(x_j, t) \left[1 + 3 c_i \cdot u(x_j, t) + \frac{9}{2} (c_i \cdot u(x_j, t))^2 - \frac{3}{2} ||u(x_j, t)||^2 \right] \tag{7a}$$

$$w = \begin{pmatrix} \frac{4}{9} & \frac{1}{9} & \frac{1}{9} & \frac{1}{9} & \frac{1}{9} & \frac{1}{36} & \frac{1}{36} & \frac{1}{36} & \frac{1}{36} \end{pmatrix} \tag{7b}$$

2.3 Boundary Handling

Equations (5), (6) and (7a) from the previous section are enough to implement a basic Lattice Boltzmann scheme with streaming and collision step. Until now, the scheme implies periodic boundary conditions (PBCs). In the 2D case this would correspond to the surface of a torus. This section briefly discusses three different boundary conditions: rigid wall, moving wall and PBCs with a pressure gradient.

In principle there are two kinds of boundary conditions for LBM [5]:

- dry-node: boundaries are located between nodes
- wet-node: boundaries are placed on lattice nodes

In the following sections only dry-nodes are considered, since they are easier to implement and retain a second order accuracy as long as the physical wall is placed exactly halfway between the nodes [5].

2.3.1 Rigid Wall

A rigid wall applies a bounce-back boundary condition[5]. This is modelled by simply reflecting the populations i which hit the wall at time t into opposite channels $\bar{i}$ at time $t + \Delta t$ as illustrated in Figure 2 [5]:

$$f_{\bar{i}}(x_b, t + \Delta t) = f_i^*(x_b, t) \tag{8}$$

The rigid wall boundary condition is applied post-streaming but requires the pre-streaming populations f^*. x_b denotes a boundary node, since non-boundary nodes are obviously not updated.

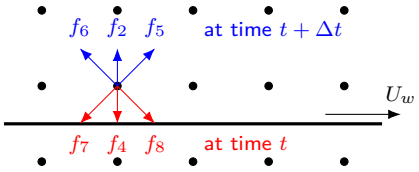


Figure 2: Schematic illustration of a bounce-back boundary condition with dry-nodes. For $U_w = 0$ this represents a rigid wall, otherwise a moving wall.

2.3.2 Moving Wall

The moving wall is similar to the rigid wall, but additionally the populations will lose or gain velocity during interaction with the wall with is proportional to the wall velocity U_w [5]:

$$f_{\bar{i}}(x_b, t + \Delta t) = f_i^*(x_b, t) - 2\omega_i \rho_w \frac{c_i \cdot U_w}{c_s^2} \tag{9}$$

Here ρ_w denotes the density at the wall, that needs to be extrapolated from the bulk, see e.g. [8] Equation (19). A simple approximation can be made by assuming it is equal to the average density. Note that we only consider walls which move parallel to their direction as shown in Figure 2.

2.3.3 Periodic Boundary Conditions with Pressure Gradient

PBCs can also be applied with a pressure variation $\Delta p = p_i n - p_o ut$ between the inlet and outlet. This is e.g. the typical case for the flow in a pipe. For this, the following conditions need to be applied for a periodicity length $L = N\Delta x$ [5]:

$$p(x,t) = p(x+L,t) + \Delta p \tag{10a}$$

$$u(x,t) = u(x+L,t) \tag{10b}$$

Since in the LBM the pressure is directly related to the density through the ideal gas equation of state, $p = c_s^2 \rho$ applies, where $c_s = \sqrt{\frac{1}{3}}$ is the speed of sound in lattice dimensions [5]. Therefore, we can apply a density gradient rather than a pressure gradient, which is easier, since the density is directly related to f (see Equation (5a)).

The boundary conditions are applied to the PDF in the following way [5]:

$$f_i^*(x_0,t) = f_i^{eq}(\rho_i n, u_N) + (f_i^*(x_N,t) - f_i^{eq}(x_N,t)) \tag{11a}$$

$$f_i^*(x_{N+1},t) = f_i^{eq}(\rho_o ut, u_N) + (f_i^*(x_1,t) - f_i^{eq}(x_1,t)) \tag{11b}$$

This requires some extra nodes at the inlet (x_0) and outlet (x_{N+1}) outside the physical simulation domain. The extra node x_0, which is still in the physical domain of the previous pipe, is related to the last node x_N or the simulation domain. The same applies to x_1 and x_{N+1}.

Note that contrary to the rigid and moving wall, PBCs with pressure gradient have to be applied pre-streaming.

3 Implementation

In this chapter, the concrete implementation of the LBM using Python and the parallelization using Message Passing Interface (MPI) are discussed.

3.1 Lattice Representation

The physical lattice domain is represented using a D2Q9 discretization and with the y-dimension being the first (index 0) and the x-dimension being the second (index 1) axis. The origin is considered to be in the upper left corner as indicated in Figure 1b.

The main quantity representing the state of the simulation is the discretized probability density function (PDF) containing the occupation numbers of all channels for all lattice points. The respective variable is called `pdf_cij` and holds a single contiguous NumPy array of shape $(9, L_y, L_x)$ where L_x and L_y are the respective lattice dimensions. The suffix of the variable name indicates the number of array dimensions as well as their meaning: `c` is the channel index, `i` and `j` are the lattice position in y- and x-direction.

Similarly, the array for the density and velocity are named `density_ij` and `velocity_aij`, where `a` denotes the Cartesian axis (x or y). They are of shape (L_y, L_x) and $(2, L_y, L_x)$ respectively.

3.2 Simulation Step

A single simulation step consists of streaming, collision, boundary handling and communication as listed in Listing 1.

```
def step():
    communicate(pdf_cij)                # see Listing 5
    boundary_handling(pdf_cij, True)    # see Listing 4
    streaming_step(pdf_cij)             # see Listing 2
    boundary_handling(pdf_cij, False)   # see Listing 4
    collision_step(pdf_cij, omega)      # see Listing 3
```

Listing 1: Single LBM Step

Details on each step are elaborated in the following sections. Note that the communication step is only required in case the simulation runs in parallel.

3.2.1 Streaming

Streaming can be implemented using the `numpy.roll` function as shown in Listing 2. This function shifts the array elements along the specified axes by the distance given by the `channel_ca` array. For example, a roll operation for channel 1 along axis 0 will assign $f_{1,i,j+1} \leftarrow f_{1,i,j}$. This is done once for all 9 velocity channels with respect to both Cartesian axes. The `channel_ca` array implements the D2Q9 discretization as given by Figure 1a.

Thereby, `numpy.roll` automatically applies periodic boundary conditions.

```python
import numpy as np

# [Y, X] shift (note: grid origin is in the top-left corner)
channel_ca = np.array([[0, 0],    # center
                       [0, 1],    # east
                       [-1, 0],   # north
                       [0, -1],   # west
                       [1, 0],    # south
                       [-1, 1],   # north-east
                       [-1, -1],  # north-west
                       [1, -1],   # south-west
                       [1, 1]])   # south-east

def streaming_step(pdf_cij: np.ndarray) -> None:
    # for each channel, shift into the corresponding direction
    for i in range(9):
        pdf_cij[i] = np.roll(pdf_cij[i], shift=channel_ca[i], axis=(0, 1))
```

Listing 2: Streaming Operation

3.2.2 Collision

The collision part from Equation (6) can easily be implemented by translating the velocity update into Einstein notation:

$$u_{aij} = \frac{1}{\rho_{ij}} c_{ca} f_{cij} \tag{12}$$

With this, the velocity update can be implemented using the `numpy.einsum` function, which implements the Einstein notation. This provides the advantage of avoiding costly Python loops and using vectorized NumPy code instead.

Listing 3 shows a simple implementation of the collision step, which first performs a density and velocity update, then calculates the equilibrium PDF f^{eq} according to Equation (7a) and finally applies the collision step with a given relaxation rate ω.

```python
import numpy as np

weight_c = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])

def equilibrium(velocity_aij: np.ndarray, density_ij: np.ndarray) -> np.ndarray:
    uc_cij = (velocity_aij.T @ channel_ca.T).T
    uu_ij = np.linalg.norm(velocity_aij, axis=0) ** 2
    return weight_c[..., np.newaxis, np.newaxis] * density_ij[np.newaxis, ...]\
        * (1 + 3*uc_cij + 4.5*uc_cij**2 - 1.5*uu_ij[np.newaxis, ...])

def collision_step(pdf_cij: np.ndarray, omega: float) -> np.ndarray:
    density_ij = np.sum(pdf_cij, axis=0)
    velocity_aij = np.einsum("cij,ca->aij", pdf_cij, channel_ca) / density_ij
    pdf_cij += omega * (equilibrium(velocity_aij, density_ij) - pdf_cij)
```

Listing 3: Collision Operation

3.2.3 Boundary Conditions

In Listing 1, the boundary handling gets applied twice, once for the pre-streaming (periodic boundary condition with pressure gradient) boundaries and once for the post-streaming boundaries (rigid or moving wall).

As shown in Listing 4, the first call creates a copy of the PDF, since the pre-streaming PDF (`pdf_pre_cij`) is required in order to apply the boundary conditions. The `boundaries` variable holds a dictionary of lists of boundary conditions that are then applied consecutively. This allows for a configurable set of boundaries, that each share a common interface like an `apply()` method, a boolean `pre_streaming` property or the list of lattice sides (left, right, top, bottom) they should be applied to.

```python
import numpy as nptexcomments

def boundary_handling(pdf_cij: np.ndarray, pre_streaming: bool) -> None:
    global pdf_pre_cij
    if before_streaming:
        pdf_pre_cij = pdf_cij.copy()
    for b in boundaries[before_streaming]:
        b.apply(pdf_cij, pdf_pre_cij)  # see Equations (8), (9) and (11)
```

Listing 4: Boundary Conditions

The exact implementations of each type of boundary conditions follow the equations in Section 2.3. Since this mainly involves copying of some PDF values, the exact code is omitted here for the sake of brevity.

3.3 Parallelization

In theory, the PDF allows for two possible approaches: parallelize in velocity direction space or the spatial domain. The first one is not a good choice, since there are only 9 channels and therefore the parallelism would be limited. Also, it would require communication during the collision step. Therefore, the second approach of a spatial decomposition is taken, which allows parallelization with much more processes.

This spatial decomposition requires communication before the streaming step, since some occupation numbers get moved past the subdomain boundaries. For this purpose, each subdomain is padded with additional ghost cells for storing a copy of adjacent values from neighboring subdomains. Figure 3 illustrates this for a 3×2 decomposition. Each subdomain has to communicate four times with its neighbors: to the right and bottom plus their reverse counterparts to left and top.

Implementation of the inter-process communication is done using mpi4py bindings for the MPI library. The respective code is outlined in Listing 5. It uses a Cartesian communicator to represent the $X \times Y$ subdomains. The communicator is instantiated with `periods=(False, False)`, since for the lid-driven cavity bounce-back boundary conditions are applied to the outer domain bound-

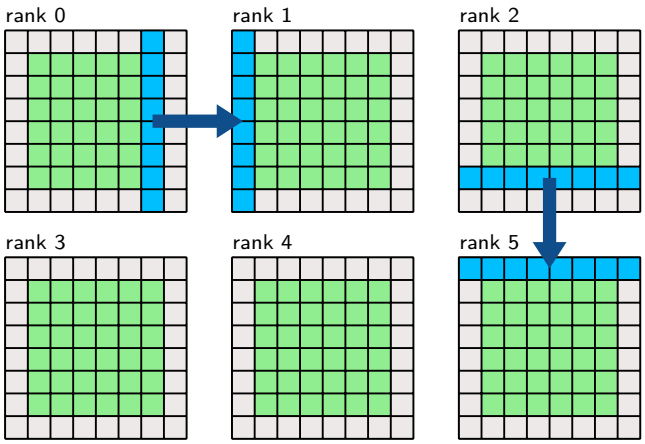


Figure 3: Domain decomposition and communication strategy. Green lattice cells are the physical simulation domain, gray cells are ghost cells for communication. In each step, the outermost physical cells are communicated to the adjacent ghost cells as indicated by the blue cells and arrows. Communication to the left and up work likewise.

```python
import mpi4py.MPI as mpi
import numpy as np

# initialize communicator
comm = mpi.COMM_WORLD.Create_cart(dims=(Y, X), periods=(False, False))

# get ranks of neighboring subdomains
# (s/d: source/destination, l/r/d/u: left/right/down/up)
sl, dl = comm.Shift(1, -1)
sr, dr = comm.Shift(1, 1)
sd, dd = comm.Shift(0, 1)
su, du = comm.Shift(0, -1)

def communicate(pdf_cij):
    # send left
    sendbuf = np.ascontiguousarray(pdf_cij[:, :, 1])
    recvbuf = pdf_cij[:, :, -1].copy()
    comm.Sendrecv(sendbuf, dl, recvbuf=recvbuf, source=sl)
    pdf_cij[:, :, -1] = recvbuf
    # send right, down, up (abbreviated)
    comm.Sendrecv(pdf_cij[:, :, -2], dr, recvbuf=pdf_cij[:, :, 0], source=sr)
    comm.Sendrecv(pdf_cij[:, -2, :], dd, recvbuf=pdf_cij[:, 0, :], source=sd)
    comm.Sendrecv(pdf_cij[:, 1, :], du, recvbuf=pdf_cij[:, -1, :], source=su)
```

Listing 5: Communication Operation

aries. Therefore, there is no need to communicate the outer boundaries of the edge domains. Respectively, introducing ghost cells only for the interior domain boundaries, but not the outer boundaries of edge domains, is sufficient. Furthermore, the boundary conditions are only applied to the concerned edge domains, because they would get overridden within interior edge domains in the next communication step anyway.

In each simulation step, one `Sendrecv` call for all four direction is made, which simultaneously sends data in one direction and receives it from the opposite direction.

Since mpi4py requires that the transferred data is contiguous in memory, both send and receive buffers are cast into contiguous arrays using either the `numpy.ascontiguousarray` function or an explicit copy operation as demonstrated for the first `Sendrecv` call. For brevity, this was omitted from the other three `Sendrecv` calls.

4 Experiments

This chapter presents the results obtained from various experiments performed with the implementation introduced in the previous chapter in order to validate the correctness of the implementation and test its scaling behavior.

4.1 Shear Wave Decay

Shear wave decay describes the time evolution of a velocity or density perturbation, which converges to zero due to the effects of viscosity. In order to have a known analytical solution to compare the simulation results against, the experiments use sinusoidal perturbation as initial conditions and observe how they evolve over time. These initial perturbation are given in Equations (13a) and (13b) for an experiment with a density or velocity perturbation respectively. The initial density is given by ρ_0 and ε defines the amplitude of the sine and therefore the max. perturbation.

$$\rho(x, t = 0) = \rho_0 + \varepsilon \sin\left(\frac{2\pi x}{L_x}\right) \tag{13a}$$

$$u_x(x, t = 0) = \varepsilon \sin\left(\frac{2\pi x}{L_y}\right) \tag{13b}$$

An analytical description of the expected behavior is obtained by considering the Navier-Stokes equations for incompressible fluids:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u + \frac{1}{\rho}\nabla p = \nu \nabla^2 u \tag{14}$$

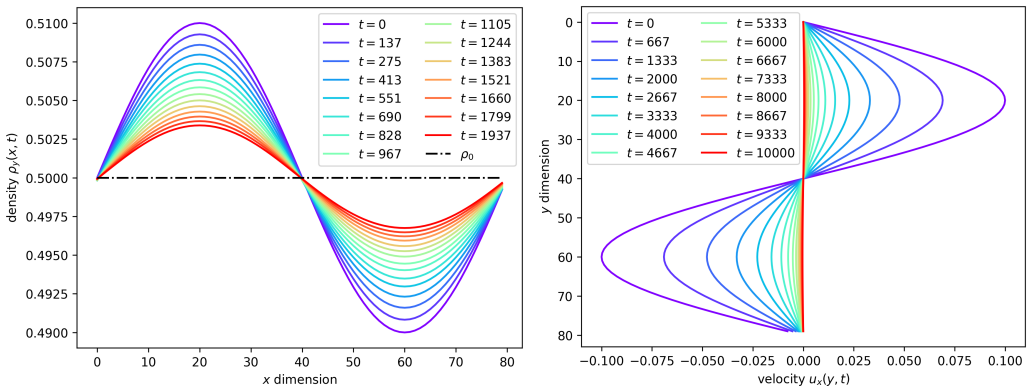
Since there is no analytical solution, we make two assumptions:

1. the perturbation is small, therefore the velocity gradient is small enough to neglect the non-linear term $(u \cdot \nabla)u$
2. the pressure gradient ∇p is small and negligible

This then yields the following analytical solution:

$$a(t) = a_0 + \varepsilon \exp\left(-\nu \left(\frac{2\pi}{L}\right)^2 t\right) \tag{15}$$

It describes a decaying exponential governed by the viscosity ν and the initial perturbation ε. The additional offset a_0 corresponds to ρ_0 in case of a sinusoidal density perturbation and is 0 for the sinusoidal velocity perturbation. Figures 4 to 6 show how the perturbation for both the sinusoidal density and velocity evolve over time and how they compare to the analytical solution given by Equation (15).

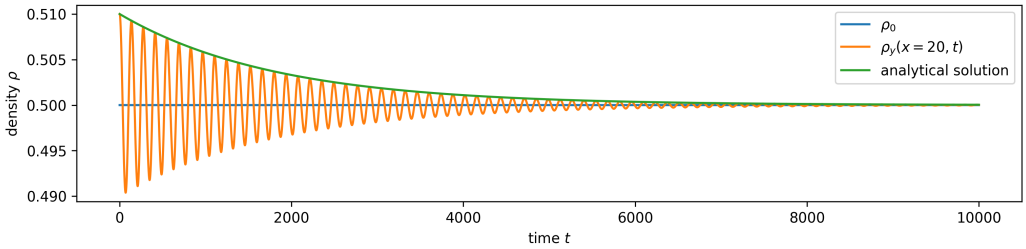


(a) Sinusoidal density **(b)** Sinusoidal velocity

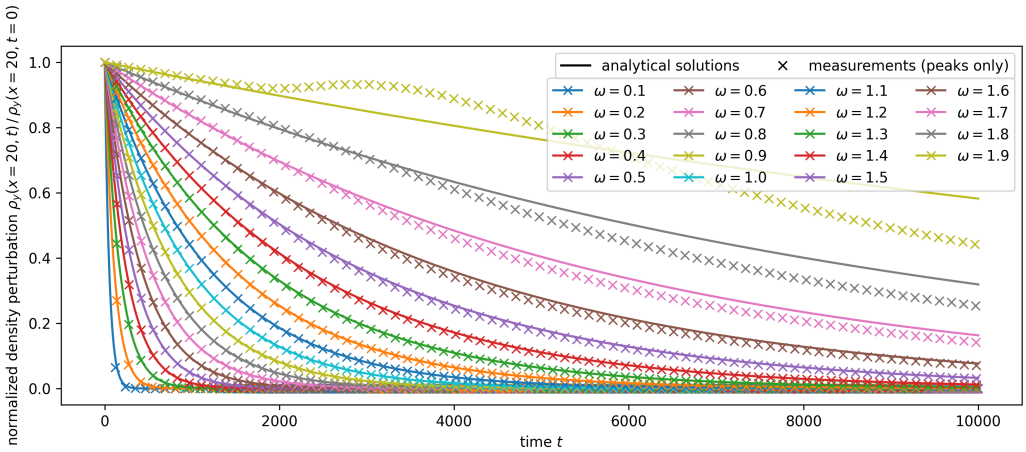
Figure 4: Shear Wave Decay: Evolution of the perturbation sine wave. The parameters for (a) and (b) are the same as in Figures 5b and 6 respectively for $\omega = 1.3$.

As expected, the perturbations retain their sinusoidal shapes but lose amplitude over time, compare Figure 4. While for the velocity decay the sinusoidal perturbation is stationary in position, the sinusoidal density decay is not, since high-density areas are constantly flowing into low-density areas while converging to the equilibrium state. Therefore, observing the density at a fixed x-position results in an oscillating decay where only the peaks align with the analytical solution as depicted in Figure 5a. The velocity decay on the other hand completely aligns with the analytical solution as seen in Figure 6.

The measured decay can be used to determine the kinematic viscosity ν of the simulation by fitting the exponential analytical solution in Equation (15) to the measured values. In case of the oscillating density decay, only the peaks get fitted. Figure 7 shows the results of this for different relaxation rates ω and compares it to the expected analytical prediction. This expected viscosity is determined through a direct relation to the relaxation rate ω as given in Equation (16), where



(a) Oscillating decay, $\omega = 1.3$



(b) Peaks only, multiple ω

Figure 5: Shear Wave Decay: Normalized time evolution of the sinusoidal density decay for different relaxation rates ω. The measured values are given by the oscillating, sinusoidal density at $x = 20$ for a lattice of size 80×20. They are set in relation to the analytical solution given by Equation (15). While (a) shows the complete oscillating decay, (b) only shows the peaks for the sake of better depiction. The simulation was run for 10000 steps and with the coefficients $\rho_0 = 0.5$ and $\varepsilon = 0.01$ for the initial sinusoidal density as described by Equation (13a).

$c_s = \sqrt{\frac{1}{3}}$ is the speed of sound in lattice dimensions [5]:

$$\nu = c_s^2 \left(\frac{1}{\omega} - \frac{1}{2} \right) \tag{16}$$

Overall, the shear wave decay yielded the expected results as shown in Figures 4 to 7. This validates the implementation of the streaming and collision operation. However, it also showed that ω values close to 0 or 2 should be avoided. A relaxation rate close to 0 gives inaccurate viscosity values as seen in Figure 7, while relaxation rates close to 2 can lead to an instable simulation

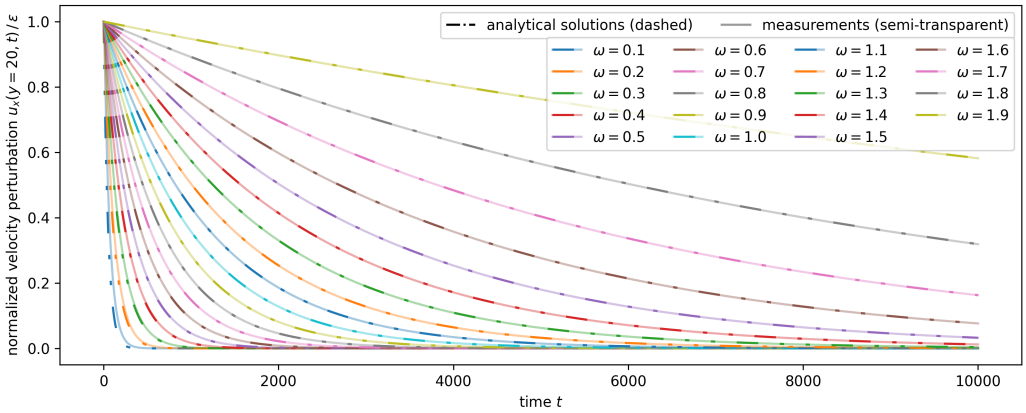
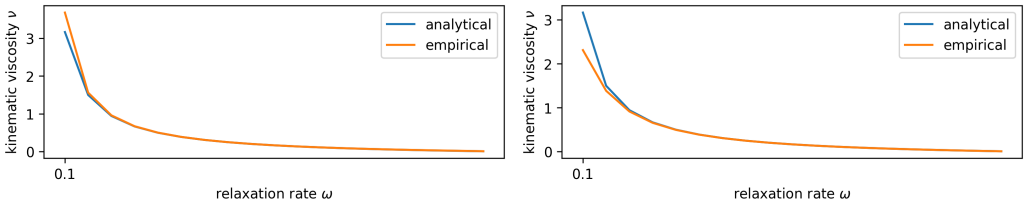


Figure 6: Shear Wave Decay: Normalized time evolution of the sinusoidal velocity decay for different relaxation rates ω. The measured values are given by the velocity at $y = 20$ in x-direction for a lattice of size 20×80. They are set in relation to the analytical solution given by Equation (15). The simulation was run for 10000 steps and with an amplitude $\varepsilon = 0.1$ for the initial sinusoidal velocity as described by Equation (13b).



(a) Sinusoidal density decay

(b) Sinusoidal velocity decay

Figure 7: Shear Wave Decay: Scaling of the kinematic viscosity ν with respect to the relaxation rate ω. The analytical solution is given by Equation (16). The empirical results are obtained by fitting (a) $||u_x||$ or (b) $||\rho - \rho_0||$ to the exponential given in Equation (15). The parameters for (a) and (b) are the same as in Figures 5b and 6 respectively.

as indicated by the large derivation from the analytical solution for $\omega = 1.9$ in Figure 5b. Even though in the case of the shear wave decay, $\omega = 1$ remained stable for more than 10000 steps and eventually also converged to the analytical solution, this may not be the case for other less artificial scenarios.

4.2 Couette Flow

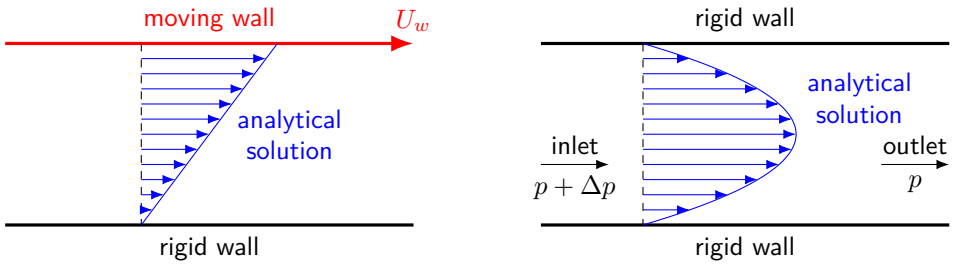
Couette flow is the flow in a viscous fluid between two boundaries of which one is rigid and one is moving with velocity U_w. In the considered case, the rigid wall is at the bottom boundary, the moving wall is at the top ($y = 0$) and moves to the right as illustrated in Figure 8a. One the inlet (left) and outlet (right), periodic boundary conditions are applied.

The moving wall induces a flow that propagates through the fluid due to the viscosity. In the steady state, a linear velocity in x-direction should be obtained for which an analytical solution is given by:

$$u_x(y) = \left(1 - \frac{y}{L_y}\right) U_w \tag{17}$$

Figure 9 shows the results of the Couette flow experiment and that both the velocity field and profile evolve as expected. In the beginning, the velocity profile has the highest values at the moving wall. Over time, the flow propagates to the bottom and eventually converges to the analytical solution.

Therefore, the obtained results validate the implementation of the moving wall boundary condition.

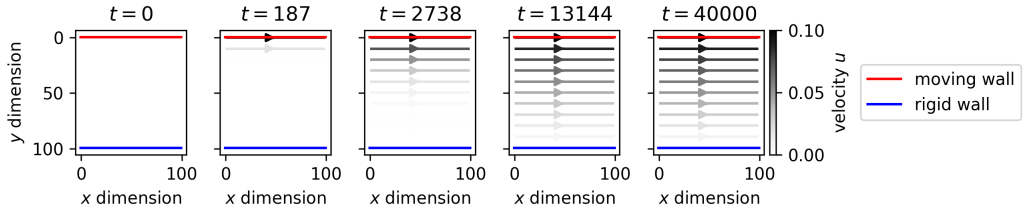


(a) Couette flow **(b)** Poiseuille flow

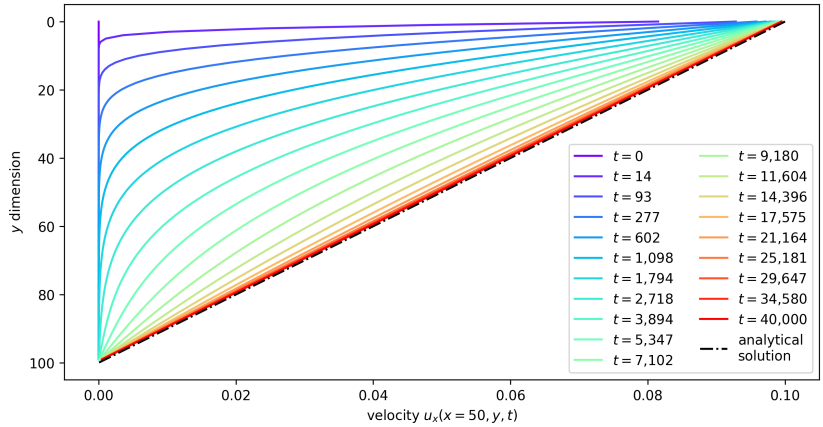
Figure 8: Conceptual visualization of the Couette and Poiseuille flows.

4.3 Poiseuille Flow

Poiseuille flow is the flow between two rigid walls as illustrated in Figure 8b. The flow is induced by a constant pressure gradient $\frac{dp}{dx}$ between the inlet and outlet.



(a) Velocity field



(b) Velocity profile

Figure 9: Couette Flow. Evolution of the velocity field and profile according to the Couette flow experiment. The results where obtained with a 100×100 lattice, an initial density $\rho(t = 0) = 1$, an initial velocity $u(t = 0) = 0$, a relaxation rate $\omega = 1.0$, a wall velocity $U_w = 0.1$ and 40000 simulation steps.

In the considered setting, the rigid walls are at the top and bottom boundaries and inlet (left) and outlet (right) are implemented using PBCs with a pressure gradient. Instead of a pressure variation Δp between inlet and outlet, a density variation $\Delta\rho$ is applied, since pressure is directly related to density through the ideal gas equation of state [5]:

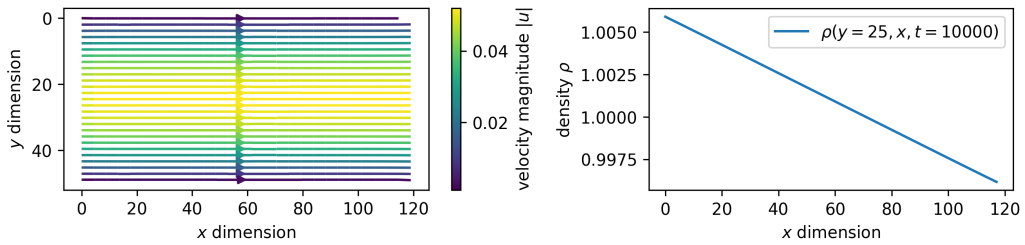
$$p = c_s^2 \rho \tag{18}$$

In the steady state, a parabolic velocity profile should be obtained, for which the analytical solution is given by [5]:

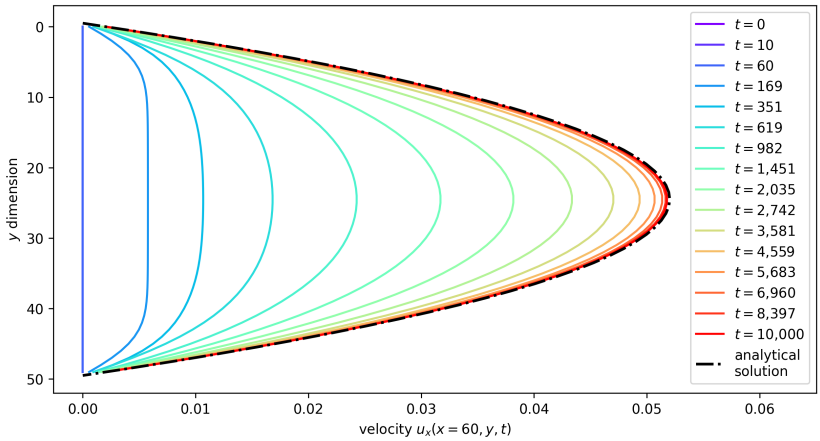
$$u_x(y) = -\frac{1}{2\rho\nu}\frac{dp}{dx} y(Y - y) \tag{19}$$

Figure 10 presents the results obtained from the Poiseuille flow experiment.

It shows how the velocity profile over time converges to the analytical solution and that a linear density gradient is obtained between the inlet and outlet. These results validate the correct implementation of the PBCs with pressure gradient.



(a) Velocity field in the steady state **(b)** Density along centerline



(c) Evolution of the velocity profile

Figure 10: Poiseuille Flow. Evolution of the velocity field and profile as well as the resulting density gradient according to the Poiseuille flow experiment. The results were obtained with a 120×50 lattice, an initial density $\rho(t = 0) = 1$, an initial velocity $u(t = 0) = 0$, a relaxation rate $\omega = 1.0$, a density gradient from $\rho_{in} = 1.005$ to $\rho_{out} = 0.995$ and 10000 simulation steps.

4.4 Lid-Driven Cavity

The lid-driven cavity is a well-known benchmark for fluid simulations [6]. It consists of a square cavity with three rigid walls and one moving wall, the sliding lid. In the considered setting, the lid is located at the top boundary and moves to the right with velocity U_w. The sliding lid induces turbulences in the cavity, characterized by the Reynolds number Re, a dimensionless number which expresses the

ratio between the inertial terms and the viscous ones and helps to predict flow patterns:

$$Re = \frac{Lu}{\nu} \tag{20}$$

In this equation L is a characteristic length, which in case of a square cavity is simply the domain size $L = L_x = L_y$, and u corresponds to the wall velocity U_w.

Figure 11 shows the steady-state velocity fields of the lid-driven cavity experiment for different Reynolds numbers and viscosity values. It demonstrates that for the same Reynolds number, the resulting flow is almost the same, despite differences in the viscosity. For example the eddy in the bottom-left corner gets more distinct with higher Reynolds numbers. A video of the time evolution of the velocity field is available in the code repository[5].

Note that the plots in Figure 11 were computed without parallelization. For a couple of cases, the resulting velocity fields of both the serial and the parallel implementation were compared. It could be observed that both were identical. Based on this, the conclusion can be drawn that the parallelization is implemented correctly.

4.5 Scaling

In order to test, how good the parallel implementation scales, the lid-driven cavity experiment was run multiple for different lattice sizes and different number of MPI processes. The scaling test were run on the bwUniCluster 2.0[6]. To compare the result, the number of million lattice updates per second (MLUPS) is used as unit. It is calculated from the lattice dimensions $L_x \times L_y$, the total number of simulation steps N_{steps} and the elapsed time T as given by the following equation:

$$\text{MLUPS} = \frac{L_x \cdot L_y \cdot N_{steps}}{10^6 \cdot T} \tag{21}$$

The results are displayed in Figure 12. It can be observed that for a few processes, the implementation scales linearly. This diminishes as the number of processes grows and therefore the size of each subdomain shrinks. An intuition for

[5] https://github.com/petwu/hpc-with-python
[6] https://wiki.bwhpc.de/e/Category:BwUniCluster_2.0

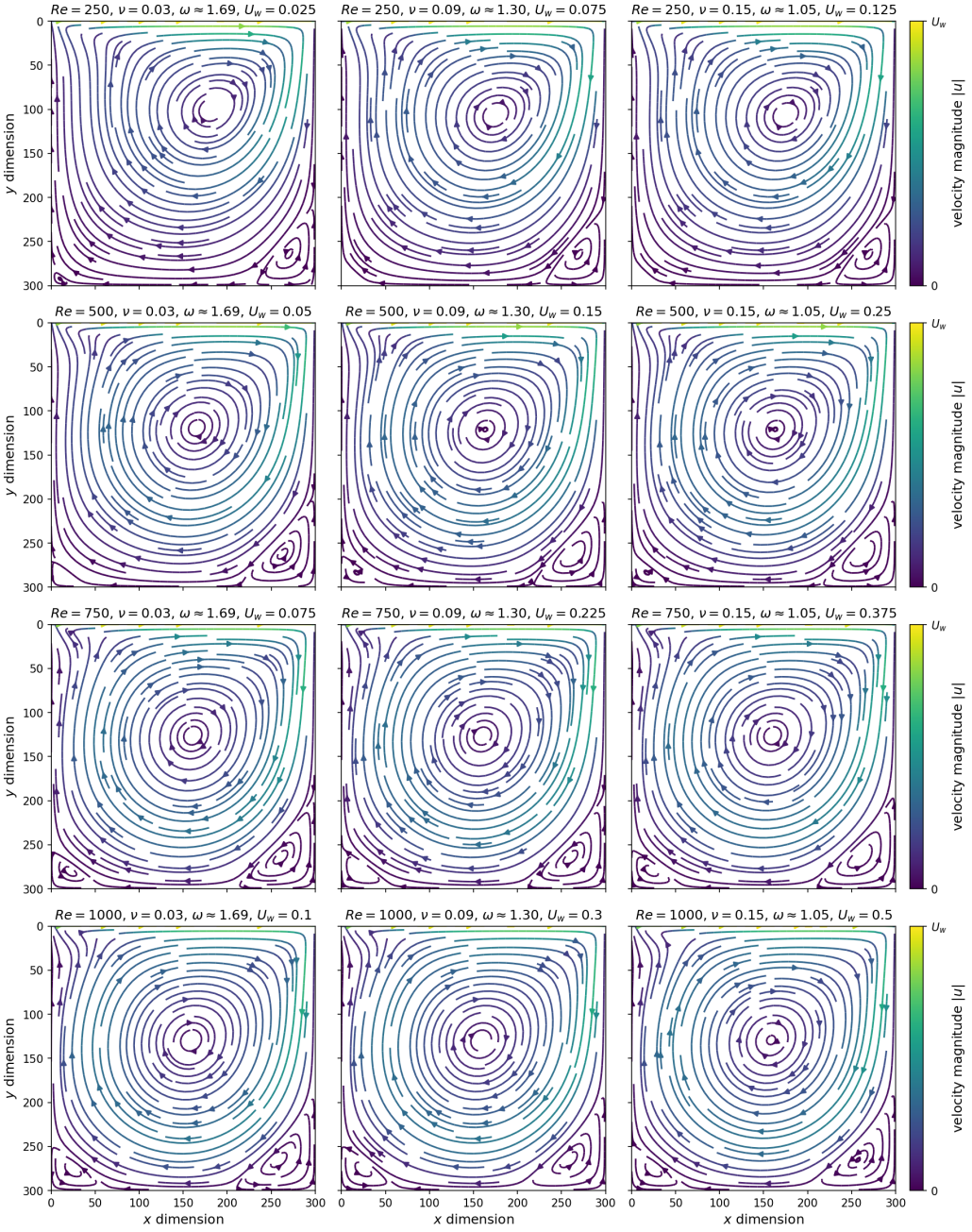


Figure 11: Lid-Driven Cavity: Streamline plots for different Reynolds numbers. Each simulation was run with a 300×300 lattice and a given set of combinations of different Reynolds numbers Re and viscosity values ν for 100000 steps. Density and velocity were initialized with $\rho(t = 0) = 1$ and $u(t = 0) = 0$. The relaxation rate ω and wall velocity U_w are deduced values, following Equations (16) and (20).

the observed behavior is given by Amdahl's Law [9]. The non-optimal speedup is caused by the necessary synchronization between processes imposed by the communication step. Decreasing of the speedup starts when the perimeter-to-area ratio grows large enough, roughly > 0.1, and hence the additional cost of the communication eventually outweighs the benefit gained by the parallelization.

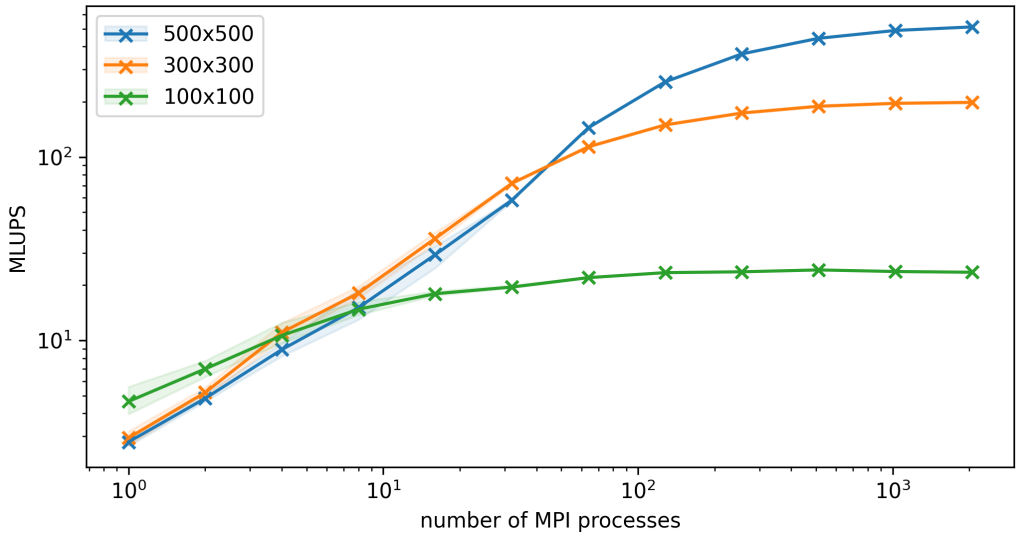


Figure 12: Scaling behavior of the lid-driven cavity simulation. The number of MPI processes was chosen to be from the set $\{2^x \mid 0 \leq x \leq 12,\ x \in \mathbb{N}\}$. Each test case was run three times. The crosses represent the mean value for each test case, while the light regions around the curve indicate the minimum and maximum values. All simulations were run with a relaxation rate $\omega = 1.7$, a wall velocity $U_w = 0.1$ and a total of 100000 steps. The million lattice updates per second (MLUPS) number is calculated from the measured elapsed time using Equation (21).

5 Conclusions

This report demonstrated a parallel implementation of the Lattice Boltzmann Method (LBM) for simulating fluid dynamics. Implementation was done in the Python programming language and the NumPy library for efficient numerical calculations. For the parallelization, the Message Passing Interface (MPI) and the respective Python bindings mpi4py were employed.

The implementation was validated using several experiments including shear wave decay, Couette flow, Poiseuille flow and lid-driven cavity. The obtained results coincide with the expected analytical solutions. Therefore, it can be assumed that the implementation is correct.

Furthermore, the scaling behavior of the parallelized implementation was tested by simulating the lid-driven cavity with different number of processes on the bwUniCluster. This showed, that the LBM has good scaling behavior. While the subdomain size handled by each MPI process is large enough, it scales linearly. Once the cost imposed by the necessary communication between neighboring subdomains outweighs the benefit gained by the parallelization, the speedup decreases and eventually stagnates.

Acknowledgements

The author acknowledges support by the state of Baden-Württemberg through bwHPC.

References

[1] Guy R. McNamara and Gianluigi Zanetti. "Use of the Boltzmann Equation to Simulate Lattice-Gas Automata". In: *Phys. Rev. Lett.* 61 (20 Nov. 1988), pp. 2332–2335. DOI: 10.1103/PhysRevLett.61.2332. URL: https://link.aps.org/doi/10.1103/PhysRevLett.61.2332.

[2] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[4] Lisandro Dalcin and Yao-Lung L. Fang. "mpi4py: Status Update After 12 Years of Development". In: *Computing in Science & Engineering* 23.4 (2021), pp. 47–54. DOI: 10.1109/mcse.2021.3083216.

[5] Krüger Timm et al. *The lattice Boltzmann method: principles and practice*. Springer: Berlin, Germany, 2016.

[6] A. A. Mohamad. *Lattice Boltzmann Method. Fundamentals and Engineering Applications with Computer Codes*. Springer London, May 2019. ISBN: 9781447174233. DOI: 10.1007/978-1-4471-7423-3. URL: https://link.springer.com/book/10.1007/978-1-4471-7423-3.

[7] P. L. Bhatnagar, E. P. Gross, and M. Krook. "A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems". In: *Phys. Rev.* 94 (3 May 1954), pp. 511–525. DOI: 10.1103/PhysRev.94.511. URL: https://link.aps.org/doi/10.1103/PhysRev.94.511.

[8] Qisu Zou and Xiaoyi He. "On pressure and velocity boundary conditions for the lattice Boltzmann BGK model". In: *Physics of fluids* 9.6 (1997), pp. 1591–1598. DOI: 10.1063/1.869307. URL: https://doi.org/10.1063/1.869307.

[9] 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*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.