High-Performance Computing with Python

Final Report

Jared Engelken

5163888

jengelken1@gmail.com

August 13, 2021

Contents

1

Introduction

High-performance computing is a growing area of study within research and the information-technology industry. The use of so-called supercomputers opens the door for faster, more efficient, and more powerful computing [1]. These computers work on the basic principle of distributing each processing job across multiple processors through parallelization. Parallelization is the method of partitioning a computing job into smaller tasks, which can each be run in parallel to each other simultaneously on multiple processors. This effectively decreases the computation time significantly, especially for CPU intensive jobs.

In order to take advantage of the parallelization hardware, appropriate algorithms have to be developed, which is generally done on a problem-specific basis [1]. Since the supercomputer splits the job into smaller tasks, the program itself must be capable of being partitioned into independent tasks, whose results can then be recombined in a simple way. Ideally, the job itself would be predisposed to this parallelization. A rather robust and detailed simulation that can be used to demonstrate the capabilities of parallelization for supercomputers is the simulation of fluid flow dynamics with the lattice Boltzmann method (LBM).

This report covers the lattice Boltzmann method for 2D fluid simulation within a box with a sliding lid. The algorithms and implementation methods are also covered, providing detail to the realization of the LBM and the parallelization approach. Furthermore, the results are shown and discussed with attention to the gain in efficiency achieved by parallelizing the program on a supercomputer.

2

Methods

The potential computing efficiency and power of supercomputers are impressive, but the proper methods need to be implemented to take advantage of such a system. Accordingly, the lattice Boltzmann method with a parallelization algorithm can be utilized to demonstrate the efficacy of such a computer.

2.1 The Lattice Boltzmann Method

The lattice Boltzmann method was originally developed from the lattice gas automata (LGA) method [2]. It looks to microscopically solve the Navier-Stokes equation for the dynamic flow of fluids through an abstract approach rather than through the traditionally numerical and macroscopic methods [3] by simulating fluid density on a lattice with particle streaming and collision processes. In recent years, this simplified method has emerged as an interesting alternative to other established methods for fluid flow simulations and now serves as a well-tested method of choice for researchers. This is largely due to its suitability for computational parallelization, as the lattice can be easily divided with only the local particle interactions being necessary [1].

2.1.1 The Boltzmann Transport Equation

The lattice Boltzmann method revolves around the implementation of the Boltzmann transport equation (BTE), which was introduced by Ludwig Boltzmann in 1872 [4]. Originally, the BTE was intended primarily for describing the statistical behavior of thermodynamic systems; however, in modern applications and for the purposes of this report's simulations, the equation is used to describe overall kinetic behavior. In general, three main concepts are inherent to the BTE. These are the particle probability density, streaming kinetics and collision kinetics, which together comprise the kinetic motion of the particles.

Probability Density

Although the LBM is a microscopic approach, the BTE does not arise from the analysis of individual particle locations and velocities within the entire

phase space, since the analysis of every molecule would lead to an egregious calculation. Instead, the BTE establishes a probability distribution for the position and velocity of an average particle such that for any instance in time t within the 6D phase space, the probability that a fluid particle of position $\boldsymbol{r}$ may be found within a small region $d^3\boldsymbol{r}$ with a given velocity $\boldsymbol{v}$ within the velocity space $d^3\boldsymbol{v}$ can be determined [5]. When considered together, the position space and velocity space form the 6D phase space:

$$d^3\boldsymbol{r}d^3\boldsymbol{v} = dxdydzdv_xdv_ydv_z. \tag{2.1}$$

The probability of finding a particle within a certain volume of this phase space is then given by:

$$dP = f(\vec{\boldsymbol{r}}, \vec{\boldsymbol{v}}, t)d^3\boldsymbol{r}d^3\boldsymbol{v}. \tag{2.2}$$

Therefore, the normalized probability of finding a particle within the entire phase space can be given as:

$$P = \int_{\Omega_{\vec{r}}}\int_{\Omega_{\vec{v}}} f(\vec{\boldsymbol{r}}, \vec{\boldsymbol{v}}, t)d^3\boldsymbol{r}d^3\boldsymbol{v} = 1. \tag{2.3}$$

Beyond this, if one were to further integrate over the six dimensions of the phase space, one could determine the number of particles within the given position and velocity space. In any case, for a homogeneous fluid of singular mass, the analysis of all particles within the phase space, given by f, is represented by merely a single particle probability density of position $\boldsymbol{r}$ and velocity $\boldsymbol{v}$. For this probability density, the motion given by $f(\boldsymbol{r_i}, \boldsymbol{v_i}, t)$ can be broken down into a streaming term and a collision term [3, 5]. A representative equation for this can be given as

$$\begin{aligned}
&\frac{df(\boldsymbol{r}(t), \boldsymbol{v}(t), t)}{dt} \\
&= \frac{\partial f(\boldsymbol{r}(t), \boldsymbol{v}(t), t)}{\partial t} + \frac{d\boldsymbol{r}(t)}{dt}\nabla_{\boldsymbol{r}} f(\boldsymbol{r}(t), \boldsymbol{v}(t), t) + \frac{d\boldsymbol{v}(t)}{dt}\nabla_{\boldsymbol{v}} f(\boldsymbol{r}(t), \boldsymbol{v}(t), t) \\
&= \left(\frac{\partial f(\boldsymbol{r}(t), \boldsymbol{v}(t), t)}{\partial t}\right)_{coll},
\end{aligned} \tag{2.4}$$

where the left-hand side of the equation is the streaming term, made up of the change in position and the change in velocity, and the right-hand side is the collision term [5].

Streaming

Considering a minuscule volume within the phase space for some $\boldsymbol{r_0}$ and $\boldsymbol{v_0}$ at instance t, it holds true that this probability density is subject to transport, such that at time $t + \Delta t$, $\boldsymbol{r_1} = \boldsymbol{r_0} + \Delta r$. Furthermore, if an acceleration is present, then the velocity will also experience a transport, such that for $t + \Delta t$, $\boldsymbol{v_1} = \boldsymbol{v_0} + \Delta v$. In combination, these two processes produce the streaming of the phase space density, represented by the left-hand side of Eq. (2.4) [5].

Collision

The streaming portion of the BTE assumes unperturbed transport so that the right-hand side is equivalent to zero. However, this is not the case in reality as particles are bound to collide with one another in non-laminar flow, which leads to scattering. This scattering process is assumed to be instantaneous and conservative of momentum and energy. Thus, for a probabilistic consideration, the collision portion of the BTE can be written as

$$\frac{\partial f}{\partial t} = \int\int\int w(\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v})\{f(\boldsymbol{x}, \boldsymbol{v_1}, t)f(\boldsymbol{x}, \boldsymbol{v_2}, t) - f(\boldsymbol{x}, \boldsymbol{v_3}, t)f(\boldsymbol{x}, \boldsymbol{v}, t)\}d\boldsymbol{v_1}d\boldsymbol{v_2}d\boldsymbol{v_3}, \tag{2.5}$$

where x is the location of the particle collision and $w(\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v})$ is the probability of two colliding particles with velocities $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ resulting in two outgoing particles with velocities $\boldsymbol{v_3}$ and $\boldsymbol{v}$. Furthermore,

$$w(\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v})f(\boldsymbol{x}, \boldsymbol{v_1}, t)f(\boldsymbol{x}, \boldsymbol{v_2}, t)$$

represents the rate at which the two particles create the density $f(\boldsymbol{v})$ and

$$-w(\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v})f(\boldsymbol{x}, \boldsymbol{v_3}, t)f(\boldsymbol{x}, \boldsymbol{v}, t)$$

represents the rate at which $f(\boldsymbol{v})$ is lost in the collision process, which effectively creates an inverse and pushes the system towards global equilibrium. This can be represented with the following equation:

$$\left(\frac{\partial f(\boldsymbol{x}, \boldsymbol{v})}{\partial t}\right)_{coll} = \frac{f^{eq}(\boldsymbol{x}, \boldsymbol{v}) - f(\boldsymbol{x}, \boldsymbol{v})}{\tau}, \tag{2.6}$$

where $f^{eq}(\boldsymbol{x}, \boldsymbol{v})$ represents the equilibrium distribution and $\tau = \frac{1}{\omega}$ is the rate at which the system is pushed to this equilibrium [5].

This equilibrium approximation implies that the distribution function $f(\boldsymbol{r}, \boldsymbol{v}, t)$ locally relaxes to an equilibrium distribution $f^{eq}(\boldsymbol{r}, \boldsymbol{v}, t)$. However, this equilibrium is dependent upon the local density $\rho(\boldsymbol{r})$ and the local average velocity $\boldsymbol{u}(\boldsymbol{r})$, where

$$\rho(\boldsymbol{r}) = \Sigma_i f_i \tag{2.7}$$

and

$$\boldsymbol{u}(\boldsymbol{r}) = \frac{1}{\rho(\boldsymbol{r})}\Sigma_i \boldsymbol{c_i} f_i(\boldsymbol{r}), \tag{2.8}$$

from which the equilibrium distribution function is given as

$$f_i^{eq}(\rho(\boldsymbol{r}), \boldsymbol{u}(\boldsymbol{r})) = w_i\rho(\boldsymbol{r})\left[1 + 3\boldsymbol{c_i} \times \boldsymbol{u}(\boldsymbol{r}) + \frac{9}{2}(\boldsymbol{c_i} \times \boldsymbol{u}(\boldsymbol{r}))^2 - \frac{3}{2}|\boldsymbol{u}(\boldsymbol{r})|^2\right], \tag{2.9}$$

and w_i are the relaxation parameters from Eq. (2.6) for each velocity channel, which can be defined for a D2Q9 lattice as

$$w_i = \left(\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right) \quad [5].$$

2.1.2 Discretization

Given the form of the BTE shown in Eq. (2.4), the only unknown is the probability density function itself $f(\boldsymbol{r}, \boldsymbol{v}, t)$, which is dependent upon time t, the position in real space $\boldsymbol{r}$, and the velocity $\boldsymbol{v}$ in the velocity space. To simplify this, the function can be discretized into a 2D mesh. Discretization is often represented by a lattice grid notated as DxQy, where x indicates the dimensionality of the lattice, and y is the number of discrete velocity vectors [1]. Though there are multiple discretizations, for the simulation described in this report a D2Q9 discretization scheme was chosen. Some other options include D2Q5 for 2D space or D3Q15 and D3Q19 for 3D space [3].

With this discretization, the spatial component of the BTE for a particle's motion is broken into 2D space on a lattice grid as shown in Fig. 2.1 below.

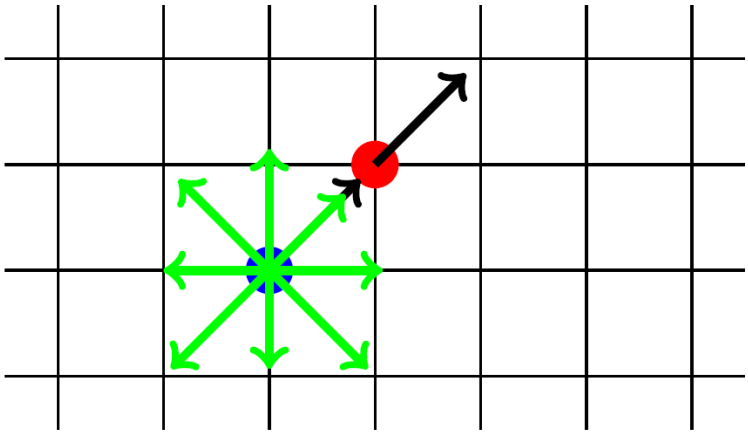

Figure 2.1: 2D Discretization of the Spatial Component $\boldsymbol{r}$ [5]

Moreover, the time component is represented through time steps Δt, and the velocity space is also discretized into nine directional channels $\boldsymbol{c_i}$ as shown below in Fig. 2.2, such that $\boldsymbol{c_i}$ can be described by the following velocity set:

$$\boldsymbol{c} = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{pmatrix}^T . \tag{2.10}$$

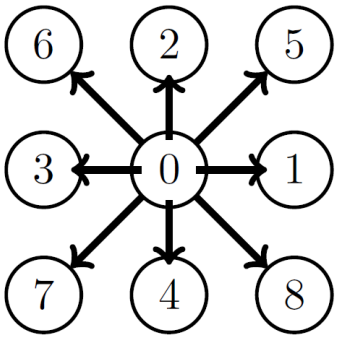

Figure 2.2: 2D Discretization of Velocity Component $\boldsymbol{v}$ in Channels $\boldsymbol{c_i}$ [5]

This allows the velocity channels shown in Fig. 2.2 to be placed on the 2D lattice shown in Fig. 2.1. Furthermore, the probability density can then transport along one of the velocity channels $\boldsymbol{c_i}$ from position $\boldsymbol{r}$ over Δt to a neighboring node $\boldsymbol{r} + \boldsymbol{c_i}\Delta t$, as shown in Fig. 2.1. Finally, with this discretization, the BTE can be simplified to [5]:

$$f_i(\boldsymbol{r} + \boldsymbol{c_i}\Delta t, t + \Delta t) = f_i(\boldsymbol{r}, t) + C_i(\boldsymbol{r}, t) \tag{2.11}$$

$$= f_i(\mathbf{r}, t) + \omega \left(f_i^{\mathrm{eq}}(\mathbf{r}, t) - f_i(\mathbf{r}, t) \right). \tag{2.12}$$

2.2 Parallelization

The simulation concept of fluid flow is simplified tremendously through the LBM, but it also presents the interesting challenge of calculating the probability density given in Eq. (2.11) and Eq. (2.12) across each lattice point i for each time step t. In serial computation, this could present a daunting task that could take quite a long time. In order to make the process more efficient, one can partition the task and delegate the computation of each partition to multiple processors running in parallel. The individual tasks would then be recombined with the aid of a message passing interface (MPI).

There are a couple methods to achieve such a parallelization. For instance, the job could be partitioned in the directional space by distributing the calculations for each direction onto separate MPI processes. However, this would not be practical because there are only nine directions for a D2Q9 discretization and parallelization would be limited to just nine processes [6]. Instead, given the inherent nature of the lattice grid in the LBM, the method used for the simulation in this report breaks down the computational job into smaller tasks by decomposing the spatial domain of the lattice.

2.2.1 The Message Passing Interface

In most modern-day computers and other electrical hardware, most central processing units (CPUs) have multiple cores that can compute in parallel. There are various hierarchical architectures that achieve parallelization such as shared memory, distributed memory, and hybrid memory, as illustrated in Fig. 2.3. Shared memory is what is often found within personal computers (PCs), where the various processing cores have shared access to a central memory bank and can implicitly pass information to each other through this memory bank. Distributed memory assigns a memory bank to each processor exclusively so that information must be passed between memory banks explicitly over a network with the assistance of a message passing interface (MPI). Supercomputers generally utilize a hybrid of these two architectures such that they are comprised of network-connected nodes, each one containing a collection of CPUs and a shared memory bank with an MPI being used to pass information between the memory banks of each node [5].

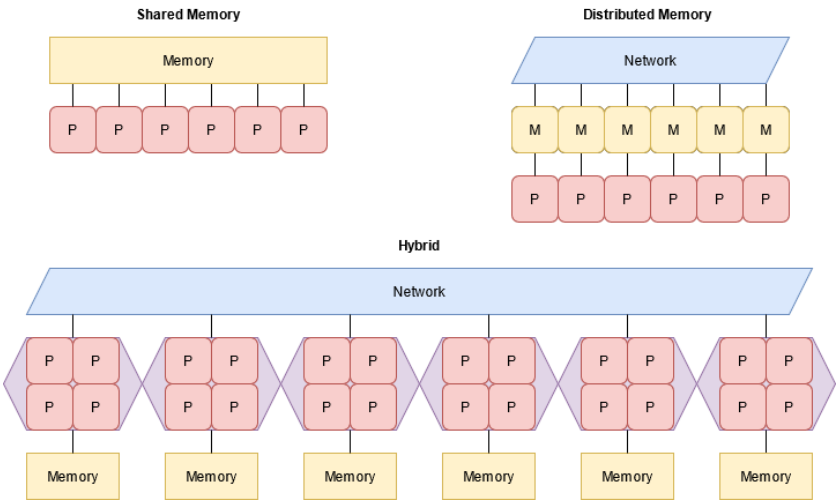

Figure 2.3: Parallel Hardware Architectures [5]

Moreover, the MPI is an application programming interface (API) and functions as an information delivery service, sending and receiving data between processors memory banks. This process of information transfer is called communication and in most cases, it is more time intensive than the actual computation. Therefore, it is important to minimize the amount of communication, even at the expense of computation [5]. This means that for any given computation, there is a threshold where the time it takes to run the task actually begins to increase with more processors. As such, for any given task and any given machine, scaling tests need to be performed in order to establish where the threshold lies.

3

Implementation - Python

In order to demonstrate the capabilities of a supercomputer with the lattice Boltzmann method and its implementation, this report covers a common fluid dynamics simulation called the sliding-lid problem. This simulation entails a box with three solid walls filled with fluid, upon which a lid slides to the side at a given rate, which in turn imparts kinetic motion within the fluid. To simulate the flow of the fluid within the box, the LBM can be utilized through a variety of approaches. For the purposes of the simulation covered in this report, the popular programming language, Python was used to execute the necessary calculations. Python is an object-oriented, high-level programming language with many additional modules available, which is ideal for the implementation of the LBM.

3.1 Modules

Current Python distributions offer basic mathematical capabilities as is, but the LBM requires the ability to manipulate matrices. As such, the python module, `NumPy` was imported. Furthermore, to provide plotting capabilities, the `matplotlib.pyplot` module was imported. Naturally, `mpi4py` was also needed for the MPI communication between parallel processes. Finally, the `time` module was imported to track the run-time of the program so as to determine the efficacy of the parallelization for different processor counts.

3.2 Approach

The assumed approach for implementing the LBM for the sliding-lid problem entails the use of a `NumPy` array, representing the lattice grid depicted in Fig. 2.1. This array is a $N \times M \times 9$ array, where N and M are the dimensions of the box's profile and the third dimension pertains to the nine phase space directions, as shown in Fig. 2.2. Beyond this, the code can essentially be separated into streaming and collision components, which are both required for establishing the box wall boundary conditions and the fluid shear wave decay. These functions are utilized in a calculation loop that recomputes the probability density in the lattice grid for each step within a given amount

of time steps. Further detail for each segment is presented below[1].

3.2.1 Streaming

If the collision term in Eq. (2.11) is temporarily ignored, the BTE simply represents a transport equation in adherence to Newton's second law of motion. In this scenario, the fluid particles move in a single direction from one lattice point to the next without deviation. Utilizing the velocity discretization of Eq. (2.10) that is depicted in Fig. 2.2, the streaming functionality can be easily coded in Python with a single line as shown below.

```
import numpy as np
c = np.array([[0,  1,  0, -1,  0,  1, -1, -1,  1],
              [0,  0,  1,  0, -1,  1,  1, -1, -1]]).T
def streaming(f):
    for i in range(1, 9):
        f[i] = np.roll(f[i], c[i], axis=(0, 1))
```

Here, the `numpy.roll()` function is used to transport the particles, or the probability densities to be more precise, by one lattice grid point in f_i for each of the nine velocity channels. An additional benefit to using the `numpy.roll()` function is that it automatically establishes periodic boundary conditions, so that mass is conserved when the particles move off the edge of the grid by respawning them on the opposite side [6].

3.2.2 Collision

As previously stated, the right-hand side of the BTE is the collision term and represents the interactions between particles. For the purposes of simplification, an equilibrium approximation is applied as described in Eq. (2.1.1) [5]. Writing the equilibrium distribution within Python is simply a matter of making a `NumPy` array with nine indices for each of the nine channels. These indices are populated with the equilibrium distribution function and the channels' respective velocities.

Once the equilibrium distribution is established, the collision term of the BTE in Eq. (2.4) can be calculated using Eq. (2.6) as shown below.

[1]Portions of the code were created with the assistance of Andreas Greiner, Jan Mees, and Lars Pastewka [5, 6, 7].

```
import numpy as np
def collision(f, omega):
    rho_coll = np.sum(f, axis=0)
    ux_coll = (f[1]-f[3]+f[5]-f[6]-f[7]+f[8])/rho_coll
    uy_coll = (f[2]-f[4]+f[5]+f[6]-f[7]-f[8])/rho_coll
    f+=omega*(equilibrium(rho_coll,ux_coll,uy_coll)-f)
    return rho_coll, ux_coll, uy_coll
```

Here, `rho_coll` is the local density $\rho(\boldsymbol{r})$ and both `ux_coll` and `uy_coll` make up the x and y components of the local average velocity $\boldsymbol{u}(\boldsymbol{r})$ as shown in Eq. (2.7) and Eq. (2.8), which are then used in the equilibrium function to solve the BTE as shown in Eq. (2.12).

It should be noted that on a more macroscopic level, these particle collisions and interactions represent fluid viscosity and the equilibrium distribution leads the fluid towards stability at large time scales. As such, when given an initial perturbation, the system will settle back to stable equilibrium over time. In the case of a sinusoidal wave perturbation, this settling process is called shear wave decay. Within Python, this can be demonstrated by simply setting an initial perturbation and pressure difference in the shape of a sine wave, as shown below, and then running the simulation calculations for equilibrium, collision, and streaming.

```
import numpy as np
epsilon = 0.01 #pressure difference
rho0 = 1.0 #rest density
x = np.arange(Nx)
wavevector = 2*np.pi/Nx
uy = np.sin(wavevector*x) #initial sine wave velocity
rho = np.ones((Nx,Ny))
for j in range(Ny): #initial sine wave pressure
    rho[:,j]= rho0 + epsilon * np.sin(wavevector*x)
```

3.2.3 Boundary Conditions

Naturally, for the sliding-lid simulation, boundary conditions need to be applied in order to simulate the solid walls of the box containing the fluid. On its own, the LBM does not account for this and the `numpy.roll()` function implemented into the streaming function imparts periodic boundary conditions on the system. This means that when fluid particles flow beyond the system boundary, they reenter on the opposite side. This is fine for mass conservation but is not realistic in terms of the simulation. Therefore, it is necessary to implement solid wall boundaries that reflect the fluid particles.

This bounce-back principle is in theory relatively easy to implement. When a streaming particle reaches the system boundary, instead of rolling it to the opposite side, the streaming direction can simply be reversed. One

method of demonstrating this is called Hagen-Poiseuille flow, which essentially depicts a laminar flow profile between two walls, such as flow within a pipe. Aside from the top and bottom boundaries, the left and right side are left open with periodic boundary conditions and a pressure difference is applied across the lattice plane so as to induce flow. Since both the top and bottom boundaries are stationary, the bounce-back function is rather simple, as shown below. Beyond that, only the pressure difference needs to be calculated.

```
def bounce_back(f):
    f_bottom = f[:,:,0].copy()
    f_top = f[:,:,-1].copy()
    #
    f[[2,5,6],:,1] = f_bottom[[4,7,8],:]
    f[4,:,-2] = f_top[2,:]
    f[8,1:-1,-2] = f_top[6,:-2]
    f[7,1:-1,-2] = f_top[5,:-2]
    return f
```

However, when implementing the sliding lid, the lid itself presents a unique problem in that the particle density and motion are also affected by the movement of the lid under no-slip conditions. In particular, the particles that collide with the sliding lid with an x velocity counter to the motion of the lid are slowed down, while those that hit the lid with a prograde velocity are sped up. However, particles that collide with the wall perpendicularly are simply reflected backward. In any case, the flow of particles between a stationary boundary and a moving boundary is known as Couette flow. Once this is properly implemented, the left and right walls can simply be included to form the final box enclosure. A Python implementation for the Couette flow is given below.

```
weights = np.array([4/9,1/9,1/9,1/9,1/9,\
    1/36,1/36,1/36,1/36])
wall_vel = 0.1 #wall speed
def bounce_back (f, wall_vel):
    f_bottom = f[:,:,0].copy()
    f_top = f[:,:,-1].copy()
    f_left = f[:,0,:].copy()
    f_right = f[:,-1,:].copy()
    streaming(f) #stream the particles
    f[[2,5,6],:,0] = f_bottom[[4,7,8],:]
    rho_wall = f_top[2]+f_top[5]+f_top[6]+f[0,:,-1]+\
    f[1,:,-1]+f[2,:,-1]+f[3,:,-1]+f[5,:,-1]+f[6,:,-1]
    f[4,:,-1] = f_top[4]
    f[7,:,-1] = f_top[5]-6*weights[7]*rho_wall*wall_vel
    f[8,:,-1] = f_top[6]+6*weights[8]*rho_wall*wall_vel
```

To include the other boundaries of the box, the respective `f[i,j,k]` would simply be set to the inverse direction for the left wall, right wall, and the four corners, as shown in the example below.

```
#left wall
f[1,0,:] = f_left[3] #west -> east
f[5,0,:] = f_left[7] #southwest -> northeast
f[8,0,:] = f_left[6] #northwest -> southeast
```

3.2.4 Parallelization

In order to run the simulation on multiple processors in parallel, it is necessary to implement an MPI. For Python, this is done with the use of the `mpi4py` module. As previously described in 2.2.1, the MPI transfers data between the processors; and in the case of the simulation portrayed in this report, the data is sections of the lattice. However, it is not possible to simply pass lattice information between processors without specifying how the lattice partitions should be combined. To solve this issue, buffer cells are assigned to the edges of the lattice partitions. These buffer cells are called ghost cells and serve merely as placeholders for any lattice data that may be received. As such, a communication function is used to define how the lattice data is transferred between partitions: either up, down, left, or right. This is done by copying the boundary data from one partition and splicing it into the corresponding ghost cell of another partition using the `Sendrecv()` function, as shown below.

```
comm = MPI.COMM_WORLD.Create_cart((ndx, ndy),
    periods=(False, False)) #ndx, ndy = np.sqrt(size)
def communicate(f_com):
    ghost = f_com[:, -1, :].copy()
    comm.Sendrecv(f_com[:, 1, :].copy(), left_dest,
            recvbuf=ghost, source=left_source)
    f_com[:, -1, :] = ghost
```

Note that the communication between processor partitions, also called ranks, is performed similarly across all ranks. This means that when the right boundary data from `rank 0` is passed to the left boundary of `rank 1`, `rank 1` also passes its right boundary data to the left boundary of `rank 2`, and so on.

4

Results

This section presents the results from the process of implementing the lattice Boltzmann method in Python and running scaling tests to measure the efficiency of a supercomputer. These results include shear wave decay comparisons for different omega values, Couette flow profiles, Poiseuille flow profiles, and images of the steady-state flow for the sliding-lid problem, as well as efficiency scaling results from the parallelization.

4.1 Shear Wave Decay

As discussed in 3.2.2, the viscosity of the fluid created by the collision interactions of the fluid particles can be quantified for different relaxation parameters through shear wave decay. Below is a graph of the analytical viscosity versus relaxation parameter.

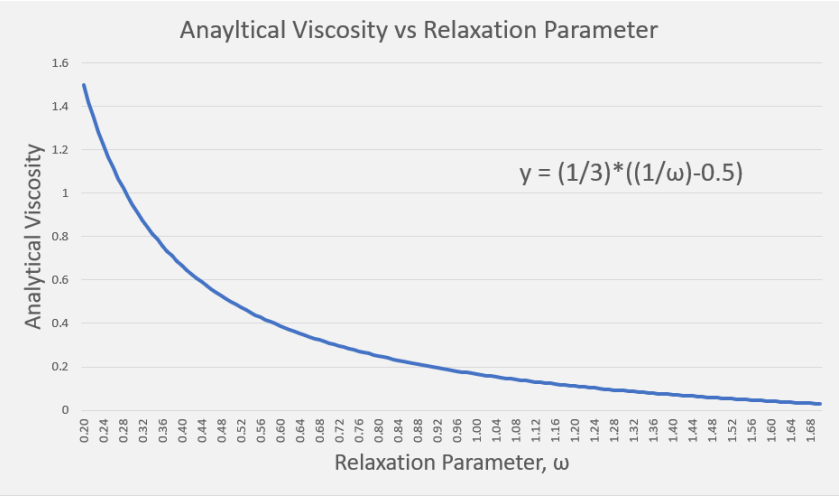

Figure 4.1: Analytical Viscosity versus Relaxation Parameter ω

Figure 4.1 shows how the viscosity logarithmically decreases as the relaxation parameter increases. In consideration of Eq. (2.6), it is clear to see how higher relaxation parameters reduce the viscosity and increase the time it takes for the fluid to reach equilibrium. Further affirmation of this is given by the shear wave decay plots for the fluid density and velocity over time.

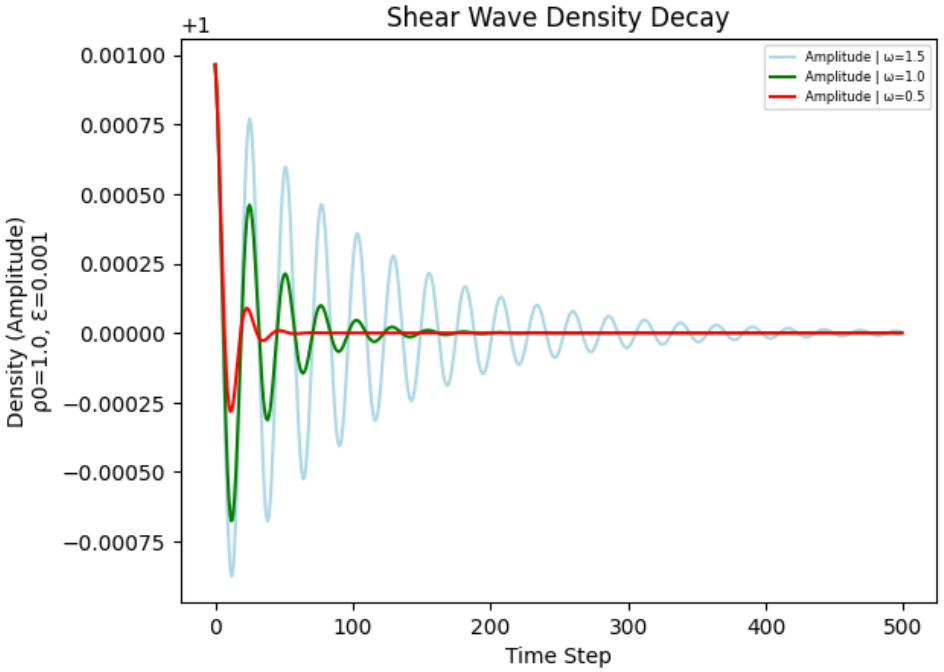

Figure 4.2: Shear Wave Density Decay for Various Relaxation Parameters $\omega = 0.5, 1.0, 1.5$, $\rho_0 = 1.0$, $\epsilon = 0.001$, size = (15,10)

Figure 4.2 depicts the decay of the fluid density as the shear wave decreases logarithmically with time. As expected from the analytical viscosity graph in Fig. 4.1, a higher relaxation parameter ω decreases the viscosity and prolongs the time it takes for the fluid to reach equilibrium.

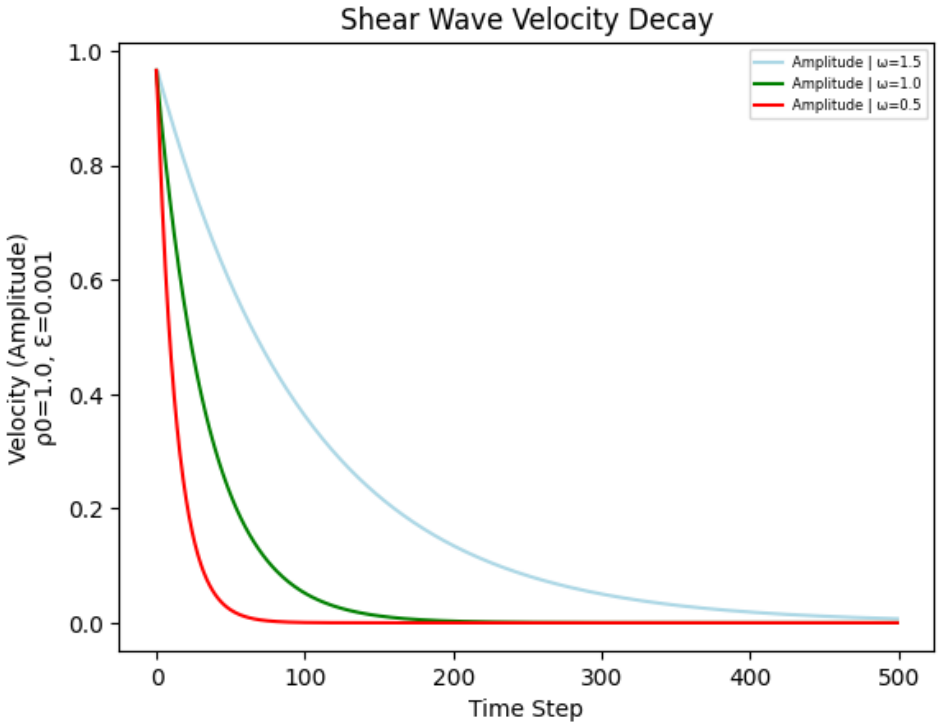

Figure 4.3: Shear Wave Velocity Decay for Various Relaxation Parameters $\omega = 0.5, 1.0, 1.5$, $\rho_0 = 1.0$, $\epsilon = 0.001$, size = (15,10)

Furthermore, the vertical particle motion established by the initial shear wave also decays logarithmically and is dependent upon the relaxation parameter ω.

4.2 Couette Flow

As stated previously, the flow of particles between a solid boundary and a moving boundary is called Couette flow. When the moving boundary, or sliding lid in this case, progresses to the side, it pulls along the fluid particles that are touching it under no-slip conditions. These particles then interact with other neighboring particles and eventually a flow profile is formed as shown in Fig. 4.4.

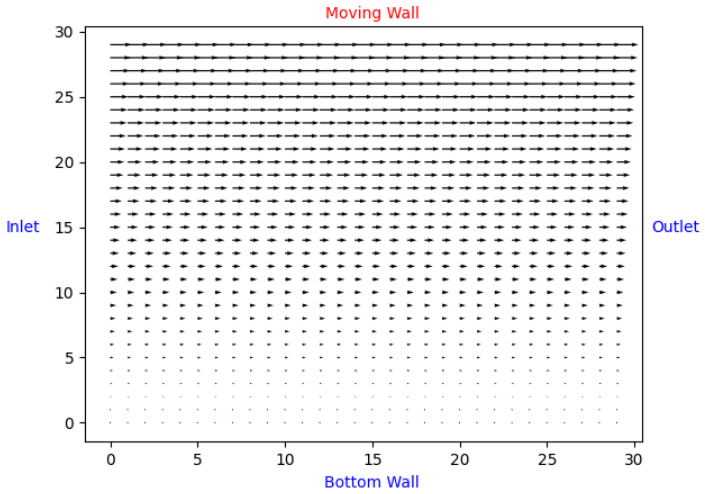

Figure 4.4: Couette Flow Velocity Profile
$\omega = 0.5$, $\rho_0 = 1.0$, wall velocity = 0.1, time step = 480, size = (30,30)

A plot of the velocity evolution can be seen below, where the velocity becomes evenly distributed over time.

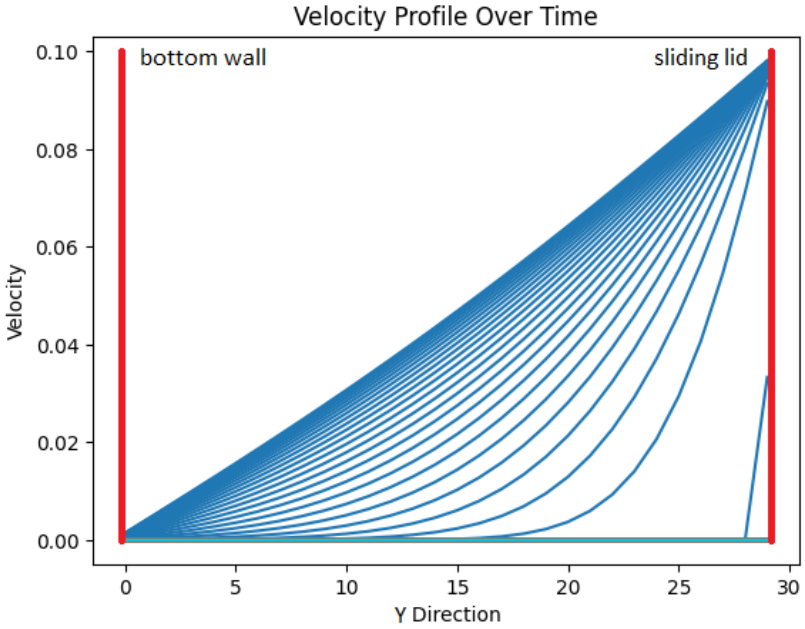

Figure 4.5: Couette Flow Velocity Profile Evolution
$\omega = 0.5$, $\rho_0 = 1.0$, wall velocity = 0.1, time steps = 500, size = (30,30)

4.3 Poiseuille Flow

Hagen-Poiseuille flow is characterized by the flow of a fluid between two stationary boundaries like within a pipe, as discussed in 3.2.3. A visual representation of this is shown below in Fig. 4.6, which displays the velocity profile over time in comparison to the analytical, steady-state velocity profile. This analytical solution is calculated from

$$u(y) = -\frac{1}{2\mu}\frac{dp}{dx}y(\mathrm{Ny} - y) \tag{4.1}$$

$$= \frac{\epsilon}{\frac{\mathrm{Nx}}{\nu}} \times \frac{1}{3}y(\mathrm{Ny} - y), \tag{4.2}$$

where Nx and Ny are the total length and width of the pipe, respectively, $\nu = \frac{1}{3} \times \left(\frac{1}{\omega} - \frac{1}{2}\right)$ is the analytical shear viscosity from 4.1, and ϵ is the pressure difference, as discussed in 3.2.3.

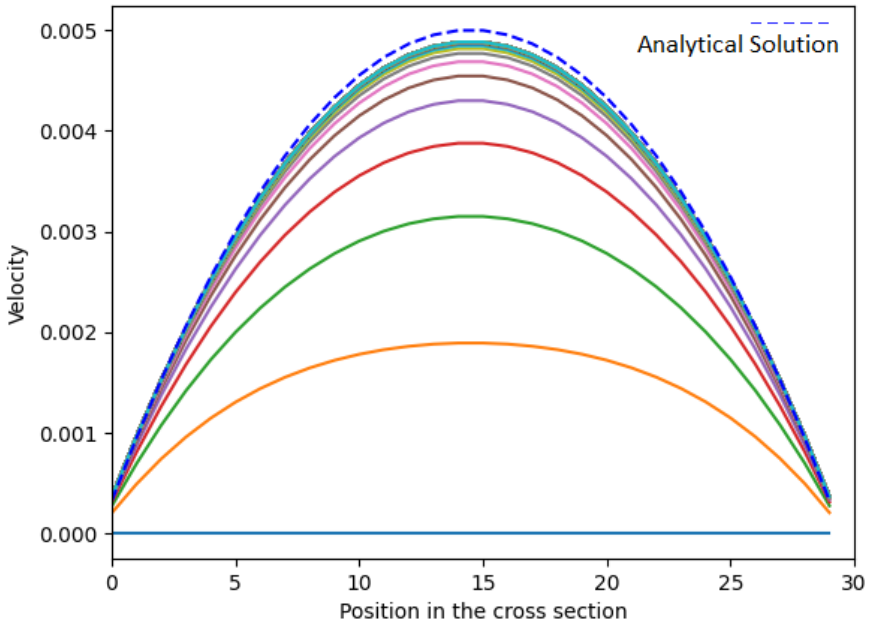

Figure 4.6: Poiseuille Flow Velocity Profile Evolution
$\omega = 0.5$, $\epsilon = 0.001$, $\rho_0 = 1.0$, time steps = 10,000, size = (30,30)

Note that despite Fig. 4.6 being oriented vertically, the flow is actually from left to right within the open-ended box.

4.4 The Sliding Lid

With all the intermediate simulations and functions established, the final lattice Boltzmann simulation for the sliding-lid problem can be created. Smaller grid sizes, like those used for the shear wave decay, Couette flow, and Poiseuille flow tests, restrict the possible amount of flow dynamics. Therefore, a larger grid size is necessary to truly show the dynamic flow progression and the eddies that form as a result of the sliding-lid's motion. Below is a snapshot of the steady-state flow in a 300x300 grid box after 50,000 time steps.

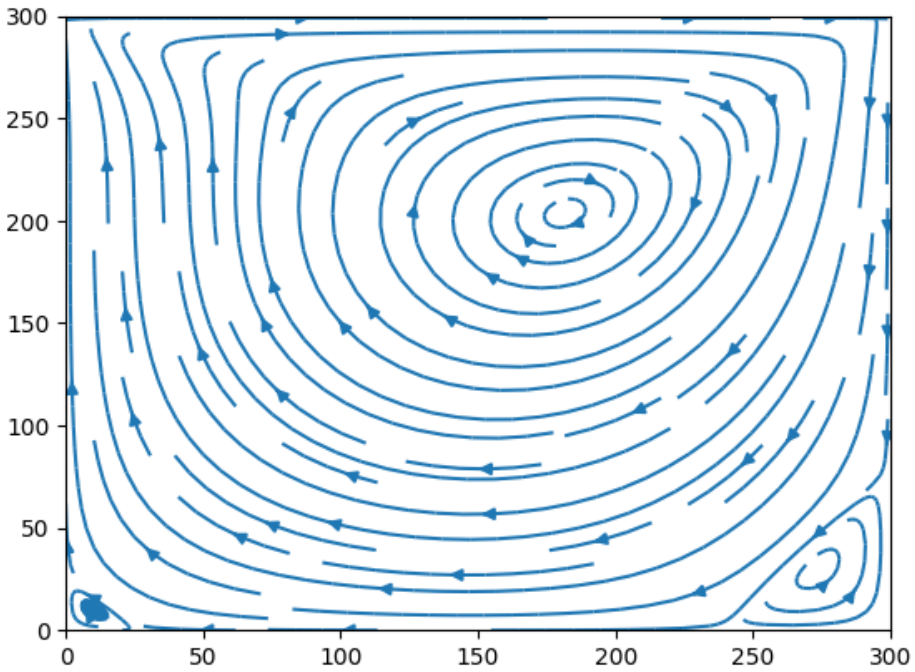

Figure 4.7: Lattice Boltzmann Sliding-Lid Simulation
$\omega = 1.0$, $\epsilon = 0.001$, wall velocity = 0.1, time step = 50000, size = (300,300)

Below is a plot of a 1,000x1,000 grid box for comparison, which was compiled on the bwUniCluster supercomputer in Karlsruhe, Germany.

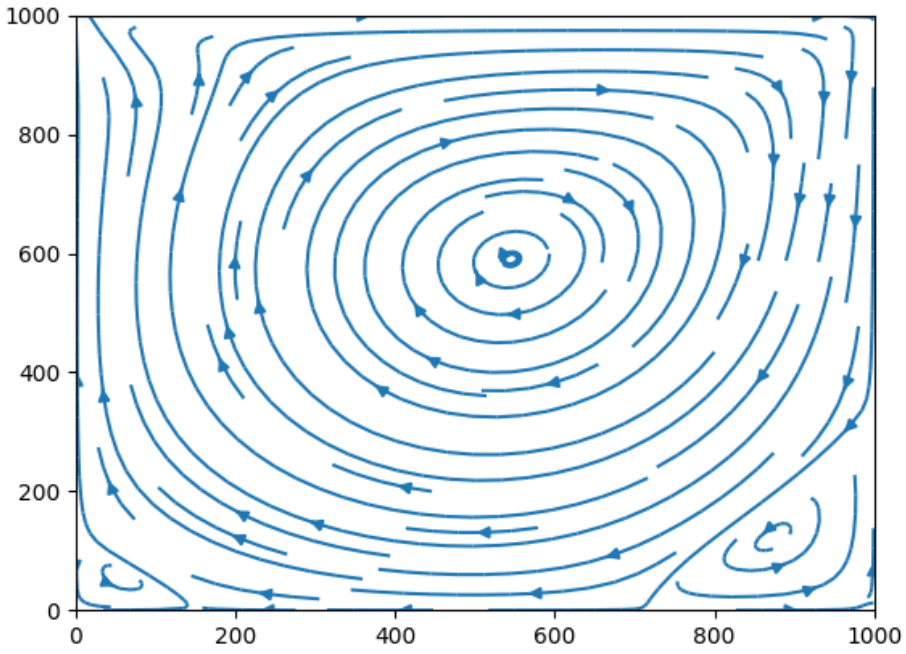

Figure 4.8: Lattice Boltzmann Sliding-Lid Simulation
$\omega = 1.0$, $\epsilon = 0.001$, wall velocity = 0.1
time step = 186000, size = (1000,1000)

4.5 Parallelization

Finally, the lattice Boltzmann simulation for the sliding-lid problem makes for a good test of the supercomputer's performance efficacy. As discussed in 2.2.1, scaling tests are necessary for determining this efficacy and for determining the threshold at which the increase in efficiency begins to falter as MPI communication time begins to outweigh computation time. A simple and quantifiable scaling test method is to measure how many million lattice updates per second (MLUPS) are achieved for a given number of MPI processes or CPUs. Below is a graph of this scaling comparison for various grid sizes computed in parallel on the bwUniCluster supercomputer.

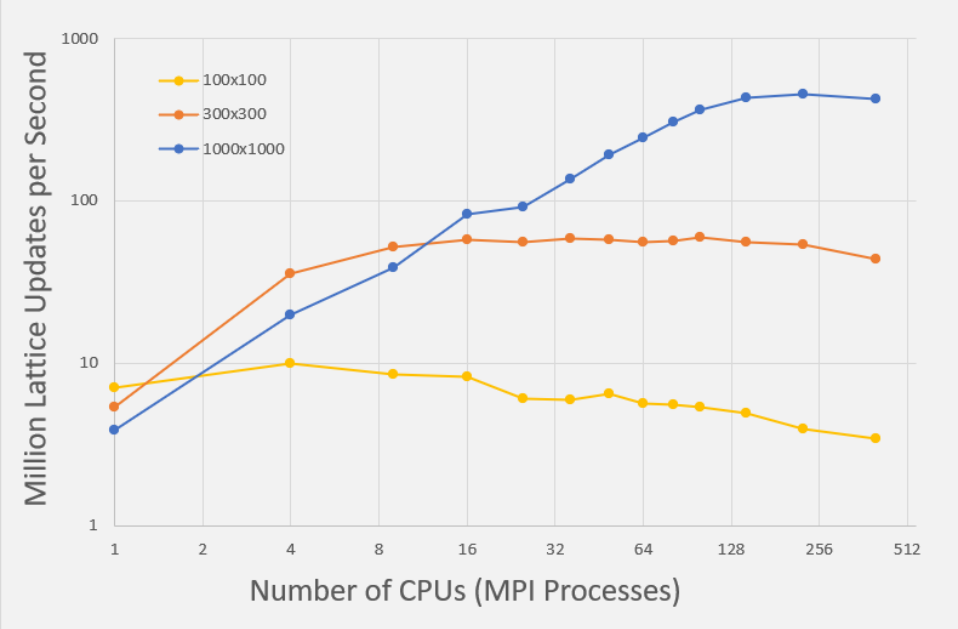

Figure 4.9: Parallel Scaling Test for Various Grid Sizes

$$\text{MLUPS} = \frac{\text{time steps}\cdot(\text{Nx}\cdot\text{Ny})}{\text{run time}}$$

5

Conclusion

In this report, the physics behind the LBM and its implementation in Python were described. The results from various flow simulations were also presented and the flow profile for the sliding-lid problem at steady-state was given. Finally, a benchmark test was performed to demonstrate the capabilities of supercomputers.

In summary, supercomputers offer an incredible amount of computing power and can be quite helpful in increasing computing efficiency. However, parallel computing is only possible for parallelizable programs, and the benefits of parallel computing come at the cost of data communication time. The threshold at which the number of CPUs begins to become more burdensome than helpful is unique for every program. As such, the sliding-lid problem implemented with the lattice Boltzmann method offers a good basis for testing parallel systems.

Bibliography

[1] M. Januszewski and M. Kostur. Sailfish: A flexible multi-gpu implementation of the lattice boltzmann method. *Computer Physics Communications*, 185(9):2350–2368, 2014.

[2] Sauro Succi. *The lattice Boltzmann equation for fluids dynamics and beyond*. Clarendon Press, 2001.

[3] D. Kandhai, A. Koponen, A.G. Hoekstra, M. Kataja, J. Timonen, and P.M.A. Sloot. Lattice-boltzmann hydrodynamics on parallel systems. *Computer Physics Communications*, 111(1):14–26, 1998.

[4] Rita G. Lerner and George L. Trigg. *Encyclopedia of physics*. VCH, 1991.

[5] Andreas Greiner, Jan Mees, and Lars Pastewka. High-performance computing: Fluid mechanics with python, 2021. University Course at Albert-Ludwigs-Universität Freiburg.

[6] Andreas Greiner and Lars Pastewka. Hpc with python: An mpi-parallel implementation of the lattice boltzmann method. In *Proceedings of the 5th bwHPC Symposium*, pages 119–133, Tübingen, 2019. Tübingen Library Publishing.

[7] Lars Pastewka. Imtek-simulation / latticeboltzmann. `https://github.com/IMTEK-Simulation/LatticeBoltzmann`, 2018.

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

[9] E. Aharonov and Daniel Rothman. Non-newtonian flow (through porous media): A lattice-boltzmann method. *Geophysical Research Letters - GEOPHYS RES LETT*, 20:679–682, 04 1993.

[10] Carlo Cercignani. *The Boltzmann Equation and its Applications*, volume 67 of *Applied Mathematical Sciences*. Springer, 1988.