High Performance Computing with Python

Final Report

Name

matricular number
mail

July 19, 2020

Contents

Abbreviations

BTE	Boltzmann Transport Equation
CI	Continuous Integration
LBM	Lattice Boltzmann Method
SIMD	Single Instruction Multiple Data
MPI	Message Passing Interface

1

Introduction

The Lattice Boltzmann Method (LBM) is a numerical, parallelizable and efficient scheme for simulating fluid flows based on the discretization of (continuous) Boltzmann Transport Equation (BTE).[1] In addition, the LBM can be extended with boundary conditions. The key property of the LBM is that it is a discrete kinetic theory approach featuring a mescoscale description of the microstructure of the fluid instead of discretizing macroscopic continuum equations. Other key advantages of the LBM include: efficient implementation by parallelization and the LBM can be applied to different kind of lattices.

We show in several two-dimensional (i.e. planar) test cases, i.e. *Couette flow*, *Poiseuille flow* and *Von Kármán's vortex street*, the correctness of our implementation as well as the significant reduction of computational time of the von Kármán's vortex street simulation by means of parallelization by spatial domain decomposition using *Python* as programming language with its highly efficient *numpy* library [2, 3] and the Message Passing Interface (MPI) [4, 5, 6].

All code is available at `https://github.com/infomon/lattice_boltzman_parallel_solver` under BSD license. We give the instructions how to reproduce the results of the experiments conducted in this report in the *README*.

Structure of report

The remainder of the report is organized as follows:

- **Chapter 2** describes the LBM. More specifically, we describe how we discretize the *Boltzmann Transport Equation (BTE)* resulting in the LBM. We also show how macroscopic quantities, e.g. density and velocity, can be calculated from the microscopic simulation. In addition, we describe several boundary conditions that can be applied in the LBM.

- **Chapter 3** describes how the LBM is implemented using *Python* as programming language. We also show how we parallelized the implementation and how we ensured software quality by unit testing.
- **Chapter 4** conducts extensive experiments showing the applicability and correctness of the implementation of the solver for the LBM.
- **Chapter 5** concludes this report.

2

Lattice Boltzmann Method

2.1 Overview

In this chapter we describe the Lattice Boltzmann Method (LBM). The main idea of the LBM is to *simulate a fluid density statistically on a lattice* instead of solving (and also discretizing) the Navier-Stokes equations.

2.2 Boltzmann Transport Equation

The Boltzmann Transport Equation (BTE) $\frac{df}{dt}$ defines the fundamental differential equation of kinematic gas theory. It describes the evolution of the probability density function $f_(\mathbf{r}, \mathbf{v}, t)$ for finding a molecule with mass m and velocity $\mathbf{v}$ at position $\mathbf{r}$ over time t. Huang [7] shows that the BTE relaxes to the Maxwell velociy distribution function. Bhatnagar et al. [8] approximate the relaxation of f towards f^{eq} as follows:

$$\frac{df(\mathbf{r}, \mathbf{v}, t)}{dt} = -\frac{f(\mathbf{r}, \mathbf{v}, t) - f^{eq}(\mathbf{v}; \rho(\mathbf{r}, \mathbf{t}), \mathbf{u}(\mathbf{r}, t), T(\mathbf{x}, t))}{\tau} \qquad (2.1)$$

where τ is the so-called characteristic time, ρ is the mass density, u is the average velocity at position $\mathbf{x}$ and T is the temperature (see section 2.3 for more details). The characteristic time determines how fast the fluid converges towards the equilibrium depending on the viscosity of the fluid. The higher the viscosity, the slower it converges towards the equilibrium. Note, that eq. 2.1 satisfies the Navier-Stokes equations.

Discretization of the BTE

The BTE of eq. 2.1 is defined in the continuous domain. In order to work with the BTE on the computer we have to discretize it in space, velocity and time. The space discretization can be done by just using a discrete lattice (e.g. two-dimensional array). To discretize the velocity and time we have to

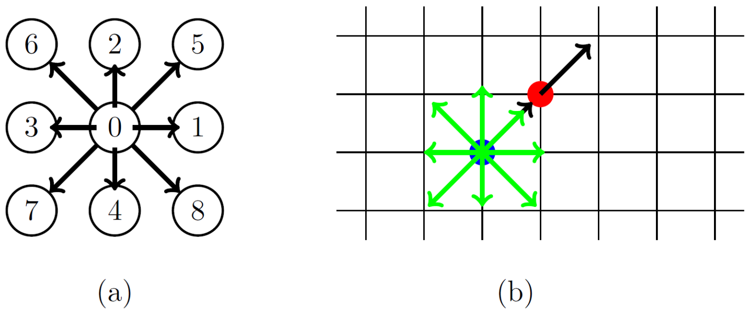

Figure 2.1: Discretization of BTE. (a) Discretization of the velocity space into nine discrete directions. The numbers 0, ..., 9 uniquely identify the direction. (b) Spatial discretization for the two-dimensional lattice. The green arrows show the possible directions of the particle in the middle. For sake of simplicity, assume that the particle only moves in direction 5. In the next time step the particle is located at the red dot.

impose that the velocity multiplied with the time is equal to some integer, i.e. the particle can only travel on the given lattice and not in-between lattice nodes.

We discretize the velocity directions with the D2Q9 scheme (see fig. 2.1a), which is two-dimensional and consists of nine discrete velocity directions. The velocity directions point to each of its neighbors in the Moore neighborhood. Note, that at the central lattice node the particle is at rest. We define the velocity vectors as follows:

$$\mathbf{c_i} = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{pmatrix}. \tag{2.2}$$

Therefore, we discretize the probability density function $f(\mathbf{r}, \mathbf{v}, t)$ to obtain the discrete probability density function $f_i(\mathbf{x}, t)$, where the subscript i indicates the direction and $\mathbf{x}$ is the discrete lattice.

Finally, we get the discretized version of eq. 2.1:

$$\underbrace{f_i(\mathbf{x} + \mathbf{c}_i\Delta t, t + \Delta t) - f_i(\mathbf{x})}_{streaming} = \underbrace{-\omega(f_i(\mathbf{x}) - f_i^{eq}(\mathbf{x}, t))}_{collision}, \tag{2.3}$$

where the *streaming* and *collision process* are the key steps in the LBM and $\omega = \frac{\Delta t}{\tau}$ is a relaxation parameter.

The equilibrium probability density function f_i^{eq} can be computed as follows:

$$f_i^{eq} = w_i\rho(x, t)(1 + 3\mathbf{c}_i\mathbf{u}(\mathbf{x}, t) + \frac{9}{2}(\mathbf{c}_i\mathbf{u}(\mathbf{x}, t))^2 - \frac{3}{2}\mathbf{u}^2(\mathbf{x}, t)), \tag{2.4}$$

where $w_i = \begin{cases} \frac{4}{9}, \ if \ i = 0 \\ \frac{1}{9}, \ if \ i = 1, 2, 3, 4 \\ \frac{1}{36}, \ if \ i = 5, 6, 7, 8 \end{cases}$. Fig. 2.1b gives a simplified example for eq. 2.3.

2.3 Moment update

In the LBM, the density ρ and velocity $\mathbf{u}$ are defined by the zeroth and first moments of the probability distribution function f, respectively:

$$\rho(\mathbf{x}, t) = \int f(\mathbf{x}, \mathbf{u}, t) \ d^3\mathbf{u}, \tag{2.5}$$

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{\rho(\mathbf{x})} \int f(\mathbf{x}, \mathbf{u}, t) \cdot \mathbf{c}(\mathbf{u}) \ d^3\mathbf{u}. \tag{2.6}$$

The discretization of those equations yields

$$\rho(\mathbf{x}) = \sum_i f_i, \tag{2.7}$$

$$\mathbf{u}(\mathbf{x}) = \sum_i f_i \mathbf{c}_i. \tag{2.8}$$

2.4 Boundary conditions

The boundary condition describes how the fluid flow behaves during streaming at the boundaries. We define the boundary node $\mathbf{x}_b$ to have at least one link to a solid or fluid node. Note, that the boundary conditions have to be placed in the correct step inside the LBM (see code listing 1). For this reason we differentiate between the *pre-streaming* probability density function f_i^* and the *post-streaming* probability density function f_i. To apply boundary conditions the probability density function after the streaming f_i is modified at each boundary node $\mathbf{x}_b$ given the pre-streaming probability density function f_i^* in each time step:

$$f_i(\mathbf{x}_b + c_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x}_b, t). \tag{2.9}$$

One question that arises is where the boundary nodes $\mathbf{x}_b$ are defined. We distinguish between so called *wet nodes* and *dry nodes* due to different domains, i.e. computational and physical domain. In the former, the computation and physical domain is the same (i.e. the boundaries are placed on the lattice nodes) but this comes with a increased difficulty for the implementation. In the latter the physical domain is half a cell away from the computational domain (i.e. the boundaries are located between the lattice nodes) retaining second order accuracy as long as the boundary is placed exactly in the middle of the lattice nodes.

Below we describe several boundary conditions. One key advantage of the LBM is its easy implementation of boundary conditions and in particular the arbitrary combination of boundary conditions as long as they do not contradict themselves.

Periodic boundary conditions

For a periodic boundary condition the flow leaving a boundary re-enters the domain on the opposite side of the domain

$$f_i(\mathbf{x}_1, t) = f_i(\mathbf{x}_N, t), \tag{2.10}$$

where $\mathbf{x}_1$ and $\mathbf{x}_N$ are the first and last node in the physical domain, respectively. Visually, we can imagine the bounce-back boundary conditions as if we have a cylindrical shape. Note, that therefore periodic boundary conditions conserve mass and momentum. The periodic boundary condition is implicitly implemented by the streaming function.

Periodic boundary conditions with pressure variation

The periodic boundary conditions with pressure variation add a density drop $\Delta\rho$ (or pressure drop Δp) between inlet and outlet. Note, that the pressure and density are related through the ideal gas of state $p = c_s^2\rho$, where c_s is the speed of sound. [9] Let's assume that we want to model a pressure drop in x-direction, then it holds $\forall y \in \{1, ..., l_y\}$ that $p(x_1, y, t) = p(x_N, y, t) + \Delta\rho$, where l_y denotes the diameter in y-direction and x_1 and x_N denote the left-most and right-most node in the LBM, respectively. Thus, we get $\rho_{out} = \frac{p_{out}}{c_s^2}$ and $\rho_{in} = \frac{p_{out}+\Delta p}{c_s^2}$, where the subscripts *in* and *out* denote the pressure values at the periodic boundaries. Note, that the velocity is the same at the periodic boundaries: $\mathbf{u}(x_1, y, t) = \mathbf{u}(x_N, y, t)$.

Let's now assume virtual nodes $\mathbf{x}_0$ and $\mathbf{x}_{N+1}$ at both ends of the periodic boundaries. Note, that the virtual nodes $\mathbf{x}_0$ and $\mathbf{x}_N$ correspond to $\mathbf{x}_N$ and $\mathbf{x}_1$, respectively. Visually we can imagine this like (infinitely) many pipes connected to each other. We decompose the probability density function into a equilibrium part f_i^{eq} and non-equilibrium part f_i^{neq}. The non-equilibrium probability density function is computed by $f_i^{neq} = f_i - f_i^{eq}$. Combining the correspondences of virtual nodes and nodes in the physical domain as well as the decomposition into (non)-equilibrium probability density function parts we obtain the inlet and outlet boundary condition, respectively

$$f_i(x_0, y, t) = f_i^{eq}(\rho_{in}, \mathbf{u_N}) + \underbrace{(f_i^*(x_N, y, t) - f_i^{eq}(x_N, y, t)\}}_{f_i^{neq}(x_N,y,t)} \tag{2.11}$$

$$f_i(x_{N+1}, y, t) = f_i^{eq}(\rho_{out}, \mathbf{u_1}) + \underbrace{(f_i^*(x_1, y, t) - f_i^{eq}(x_1, y, t)\}}_{f_i^{neq}(x_1,y,t)} \tag{2.12}$$

Bounce-back boundary

The bounce-back boundary condition applies a no-slip condition at the boundary. It simulates the interaction between the fluid with a non-moving wall without slip. It can also be applied to a stationary obstacle such as a plate.

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

where the index $\bar{i}$ denotes the conjugate channel of i, e.g. the conjugate channel of 1 is equal to 3.

Moving wall

The moving wall extends the bounce-back boundary condition by taking into account the gain or lose of momentum of particles during interaction with the moving wall. Thus, we extend the bounce-back boundary condition with an extra term for the momentum change

$$f_{\bar{i}}(\mathbf{x}_b, t + \Delta t) = f_i^*(\mathbf{x}_b, t) - 2\omega_i \rho_w \frac{\mathbf{c}_i \cdot \mathbf{u}_w}{c_s^2}, \tag{2.14}$$

where c_s is the speed of sound, ρ_w and $\mathbf{u}_w$ are the density and velocity at the wall, respectively. The velocity at the wall $\mathbf{u}_w$ is equal to $\begin{pmatrix} U_w \\ 0 \end{pmatrix}$ for a tangentially moving wall in x-direction with wall velocity U_w. There are two main options for the estimation of the density at the wall ρ_w:

1. The density at the wall ρ_w is equal to the *average* density $\bar{\rho}$.
2. The density at the wall ρ_w is *extrapolated* from the densities ρ next to the wall. Depending on the order of the extrapolation, we use more or less nodes.

Open boundary

[10] describe open boundaries consist of inlets and outlets where the flow can either enter or leave the computation domain and where we typically *impose velocity or density profiles*. We implement the inlet as follows:

$$f_i(\mathbf{x}_b, t + \Delta t) = f_i^{eq}(\rho_{in}, \mathbf{u}_{in}) \; \forall i \in \{0, ..., 8\}, \tag{2.15}$$

where ρ_{in} and $\mathbf{u}_{in}$ are the density and velocity at the inlet, respectively.

For the outlet, we implement a first-order extrapolation scheme by using the information from the second last node $\mathbf{x}_{b_2} = \mathbf{x}_b - \Delta\mathbf{x}$

$$f_i(\mathbf{x}_b, t + \Delta t) = f_i(\mathbf{x}_{b_2}, t), \tag{2.16}$$

where i denotes the indices pointing into the domain.

3

Implementation

In this chapter we will describe how we implement the algorithm using *Python* as programming language.

3.1 Overview

Code listing 1 shows the pseudocode of the iteration loop of the LBM. As input we can specify the geometry of the physical domain, the boundary conditions (see section 2.4 for more details) as well as the initial conditions.

First we initialize the density ρ and velocity $\mathbf{u}$ and compute the initial value of the probability density function $f_i^{eq} = f_i$.

Then we iterate in a loop over several steps as long as the stopping criterion (e.g. maximum time steps) is not satisfied. Note, that there is some flexibility when to apply which step. [10, 11] The following order of steps corresponds to the order in the implementation of the LBM. We first compute the equilibrium function f_i^{eq} given the current density ρ and velocity $\mathbf{u}$. In the collision step we simulate the effects of collisions between particles (see section 2.2 for more details). After that, we simulate the streaming of f_i, i.e. we simulate the movement of particles to the nearest neighbour lattice nodes using the D2Q9 discretization. Then we apply potential boundary conditions on the probability density function f_i. Note, that we first apply the streaming operation at every node (including the boundary nodes $\mathbf{x}_b$) and then correct the boundary nodes $\mathbf{x}_f$ after the streaming. This has the advantage that the implementation of the streaming is easier. Lastly, we compute the density ρ and velocity $\mathbf{u}$ (see section 2.3 for details on the formulas on how to compute the macroscopic quantities).

After running the LBM we can obtain the density ρ and velocity $\mathbf{u}$ as macroscopic quantities.

```
Input: Geometry and parameters $l$, $h$, $U$, $\nu$,...; boundary
       conditions; initial conditions
Output: Final density $\rho$ and velocity $\mathbf{u}$
1 initialize $\rho$ and $\mathbf{u}$
2 compute $f_i$ and $f_i^{eq}$
3 while stopping criterion is not satisfied do
4     compute equilibrium function $\rho,\ \mathbf{u} \rightarrow f_i^{eq}$        ▷ eq. 2.4
5     collision step $f_i^* = f_i(\mathbf{x},t) - \frac{\Delta t}{\tau}(f_i(\mathbf{x},t) - f_i^{eq}(\mathbf{x},t))$    ▷ eq. 2.3
6     streaming $f_i(\mathbf{x} + c_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x},t)$      ▷ eq. 2.3
7     apply boundary conditions $f_i(\mathbf{x}_b + c_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x}_b,t)$
        ▷ section 2.4
8     moment update $f_i \rightarrow \rho,\ \mathbf{u}$        ▷ eq. 2.7 & 2.8
9 end
```

Code listing 1: Pseudocode of the iteration loop of the LBM.

3.2 Basic implementation in Python

In this section we show how we implemented the basic equations and data structures introduced in chapter 2. We use Python as programming language and use the python libraries *numpy* [2, 3] for array operations, *scipy* [12] for some more complex scientific computations and *matplotlib* [13] for visualizing the obtained results. And key advantage of numpy is to vectorize arrays, which lowers the computational time.

We represent the (discrete) probability density function $f_i(\mathbf{x})$ as a numpy array of size $l_x \times l_y \times 9$, where l_x and l_y are the size of the lattice in x- and y-direction, respectively. The last dimension (e.g. 9) of the numpy array corresponds to the discretized velocity direction. We represent the velocity directions $\mathbf{c}$ and the weights w_i as numpy arrays with size 9×2 and 9, respectively.

In steps 4 and 5 of code listing 1 we simulate the collision part of eq. 2.3. We implement the computation of the equilibrium probability density function f_i^{eq} of eq. 2.4 and the right-hand side of eq. 2.3 with vectorized numpy code.

Code listing 2 shows the implementation of the collision step, separated into equilibrium probability density function computation and the collision.

In steps 6 of code listing 1 we simulate the streaming part of eq. 2.3. Code listing 3 shows the implementation using *np.roll*. The function rolls the the data in the direction specified by the *axis* argument. With the argument *shift* we specify the number of places elements are shifted according to the discrete velocity directions $\mathbf{c}$. Note, that the function automatically implements the periodic boundary condition.

```
Input: density $\rho$; velocity $\mathbf{u}$, relaxation parameter omega $\omega$
Output: Probability density function before streaming $f_i^*$
w_i = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])
def f_eq(rho:np.ndarray, u:np.ndarray)→np.ndarray:
    w_i = w_i[np.newaxis, np.newaxis, ...]
    ci_u = u @ c_i.T
    uu = (np.linalg.norm(u, axis=-1) ** 2)[..., np.newaxis]
    rho = rho[..., np.newaxis]
    return w_i * rho * (1 + 3 * ci_u + 9/2 * ci_u ** 2 - 3/2 * uu)
f_pre = f + (f_eq(rho, u) - f) * omega
```

Code listing 2: Python implementation of the collision step. Note that the symbol @ is a shorthand for the matrix multiplication in NumPy.

```
Input: Probability density function before streaming $f_i^*$
Output: Probability density function after streaming $f_i$
c_i = np.array([[0,0],[1,0],[0,1],[-1,0],[0,-1],[1,1],[-1,1],[-1,-1],[1,-1]])
def streaming(f_pre:np.ndarray)→np.ndarray:
    f_post = np.zeros_like(f_pre)
    for $i$ in range (9):
        f_post[..., i]=np.roll(f_pre[...,i], shift=c_i[i], axis=(0,1))
    return f_post
```

Code listing 3: Python implementation of the streaming step.

For the other boundary conditions we implement additional functions which correct the boundary nodes as described in section 2.4. Code listing 4 gives an overview of the different implementations. The implementation of the boundary conditions follows directly from the formulas. For example when we implement the bounce-back boundary condition on a non-moving wall at the bottom, we bounce back channels 4, 7 and 8. That is, we assign the pre-streaming probability density function at the boundary nodes $f_i^*(\mathbf{x}_b, t)$ to the conjugate channels (e.g. 2, 5, 6) of the post-streaming probability density function at the boundary nodes $f_i(\mathbf{x}_b, t + \Delta t)$. For the implementation of the bounce-back condition for a object inside the domain such as a vertical plate we apply the bounce-back conditions on the corresponding two columns. We have to take corner nodes into special consideration. There we just bounce-back two channels. For example consider the lattice node at the top corner on the left side of the plate. There we only bounce back channels 1 and 8. We implement the moving wall boundary condition in a similar way but with the additional term $-2\omega_i \rho_w \frac{\mathbf{c}_i \cdot \mathbf{u}_w}{c_s^2}$ to take the momentum change into account. For the density at the wall ρ_w we use the average density. For the inlet boundary condition we assign the equilibrium probability density function given the inlet density ρ_{in} and inlet velocity $\mathbf{u}_{in}$. For the outlet boundary condition we assign the second last nodes of the probability density function of the previous time step to the last nodes of the probability density function of the current time step. For the periodic boundary conditions with pressure variation we extend the domain with virtual nodes at both ends of the periodic boundaries. The implementation then is straightforward.

3.3 Parallelization

We parallelize the LBM using spatial domain decomposition and Message Passing Interface (MPI) [4, 5, 6]. MPI is a *communication protocol* for programming parallel computers, i.e. Single Instruction Multiple Data (SIMD) [14] by executing the same operation on multiple data points simultaneously.

We decompose the computational domain into sub domain using a cartesian topology. To implement this we use the MPI function `Create_cart` creating a cartesian topology. We decompose the full computational domain into sub domains of roughly equal size (see fig. 3.1 for an example). If the domain is not exactly divisible into the subdomains we extend the rightmost or topmost process with the division remainder. Note that this could lead to imbalanced subdomains, but comes with an easier transformation from global coordinates into local (i.e. process-level) coordinates. For the communication we extend the sub domains with ghost cells around the actual computational domain.

The collision step of the LBM is embarrassingly parallel, since the colli-

```
# Bounce-back
f_post[x_b, conjugate[i]]=f_pre[x_b,i]
# Moving wall
f_post[x_b, conjugate[i]]=f_pre[x_b,i] - 2· w_i[i]· avg_rho · (c_i[i]
    @ u_w) /(c_s **2)
# Inlet
f_post[0,:, i]=f_eq(rho_in,u_in)[0,:,i]
# Outlet
f_post[-1,:, i]=f_previous[-2,:,i]
# Periodic boundary condition with pressure variation
f_post[0,:,i]=f_eq(rho_in, u[-2,...])[...,i].T + (f_pre[-2,:,i]-f_eq(rho,
    u)[-2,:,i])
```

Code listing 4: Exemplar Python implementation of the boundary condition. Note that the index i refers to specific channels depending on the boundary conditions. For details see section 2.4. Note that @ and *.T* are shorthands for matrix multiplication and transpose in NumPy.

sion step operates only locally and thus requires no communication between processes.

For the streaming step we have to consider particles moving from one domain to a neighboring domain (i.e. another process) corresponding the the channel. We implement this by extending each domain by *ghost cells* around the actual computational domain. Before streaming we communicate the lattice nodes adjacent to the ghost region into the ghost points of the neighboring domain according to the specific channel.

Let's consider the example of fig. 3.1. We consider a 2×2 decomposition of the original computational domain and we denote the lower left subdomain with rank 0, upper left with rank 1, lower right with rank 2 and upper right with rank 3. W.l.o.g., we first consider the rightmost nodes $\mathbf{x}_r$ of each sub domain. We communicate the rightmost nodes $\mathbf{x}_r$ from each process to the neighboring process on the right. In our particular example this means that we communicate the rightmost nodes $\mathbf{x}_r$ of the process with rank 0 to the ghost cells on the left side of process with rank 2. Accordingly, we communicate from process 1 to process 3. Note that we do not communicate from process 2 to 0 or 3 to 1 or more generally we do not communicate outer boundaries of edge domains, since we set periods=(False, False) when calling the MPI function Create_cart. This slightly reduces the amount of required communication. We repeat this procedure for the leftmost nodes $\mathbf{x}_l$, bottommost nodes $\mathbf{x}_b$ and topmost nodes $\mathbf{x}_t$ accordingly. This totals in four communication steps. Code listing 5 shows the implementation of the communication step. We call the communication step right before the

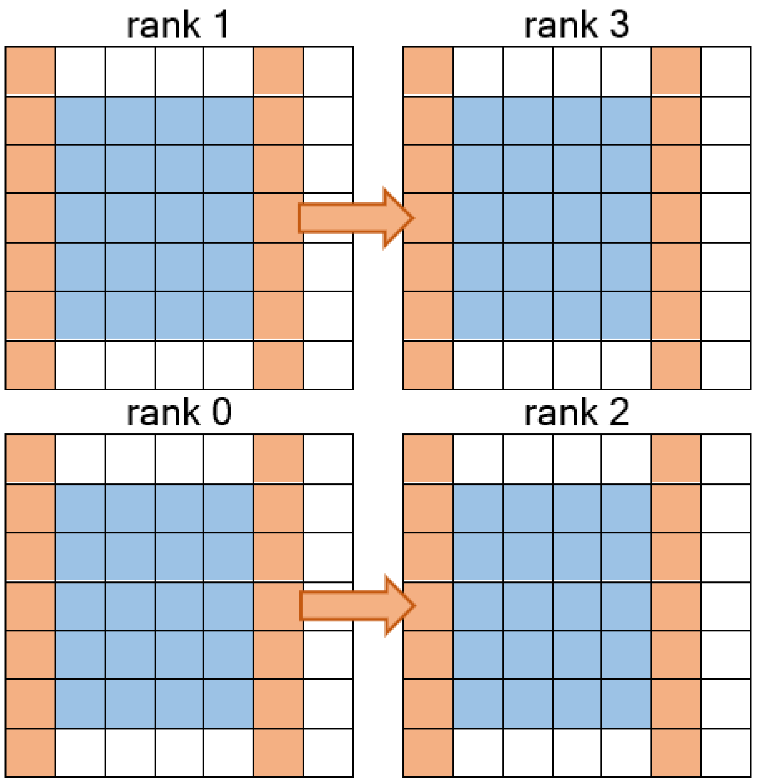

Figure 3.1: Example of spatial domain decomposition and communication strategy. We decompose the original domain into a 2×2 grid of sub domains of roughly equal size (blue lattice points). We add additional ghost nodes around the actual computational domain (white lattice points). In the communication step, we communicate the rightmost, leftmost, bottommost and topmost lattice points in the actual computational domain into the neighboring ghost lattice points. In the figure we show the communication step of the rightmost lattice points to the ghost cells of the respective right neighbors.

streaming step in code listing 1.

```
from mpi4py import MPI
from typing import Callable
def communication(comm: MPI.Intracomm)->Callable[[np.ndarray], np.ndarray]:
    left_src, left_dst = comm.Shift(direction=0, disp=-1)
    right_src, right_dst = comm.Shift(direction=0, disp=1)
    bottom_src, bottom_dst = comm.Shift(direction=1, disp=-1)
    top_src, top_dst = comm.Shift(direction=1, disp=1)
    def communicate(f: np.ndarray)->np.ndarray:
        # send to left
        recvbuf = f[-1, ...].copy()
        comm.Sendrecv(f[1, ...].copy(), left_dst, recvbuf=recvbuf,
          source=left_src)
        f[-1, ...] = recvbuf
        # send to right
        recvbuf = f[0, ...].copy()
        comm.Sendrecv(f[-2, ...].copy(), right_dst, recvbuf=recvbuf,
          source=right_src)
        f[0, ...] = recvbuf
        # send to bottom
        recvbuf = f[:, -1, :].copy()
        comm.Sendrecv(f[:, 1, :].copy(), bottom_dst,
          recvbuf=recvbuf, source=bottom_src)
        f[:, -1, :] = recvbuf
        # send to top
        recvbuf = f[:, 0, :].copy()
        comm.Sendrecv(f[:, -2, :].copy(), top_dst, recvbuf=recvbuf,
          source=top_src)
        f[:, 0, :] = recvbuf
        return f
    return communicate
```

Code listing 5: Python implementation of the communication step.

In the boundary conditions we transform global coordinates *global_coord* into local (i.e. process-level) coordinates *local_coord* in order to apply the boundary conditions at the correct global position in the lattice grid. Since we only increase the size of sub domains on the right or top, we can compute local coordinates straightforward. From the MPI function Get_coords we get the process coordinates *proc_coord* of the cartesian topology. We compute the local x coordinate $local\_coord_x$ as follows:

$$local\_coord_x = global\_coord_x - proc\_coord_x * (l_x // proc\_size_x) + 1, \quad (3.1)$$

where l_x is the lattice grid size in x direction and // denotes the integer division. Note, that we add 1, since we have to take the ghost cell in the specific process into account. We can apply computation similarly for the local x coordinate $local\_coord_y$.

3.4 Software quality

Static typing

Python is a dynamically typed language. That is that the Python interpreter does type checking only in runtime and the type of a variable is allowed to change. The opposite of dynamic typing is static typing. It is introduced by *PEP 484*[1] in Python. In static typing, the types of the variables are checked before runtime and the change of types is generally not allowed. Note, as an exception type casting is a way to change the type of a variable in many languages.

Dynamic typing allows for rapid prototyping and thus it enables fast software development. On the other side static typing can help to catch errors due to type errors, document the code and help to build a cleaner software architecture. The last point in particular ensures that the programmer thinks about the types of the variables and uses the correct types. Thus, in any larger project typing is critical to build and maintain clean code.

Unit testing

One key component of every software project is extensive testing of the software. To this end, we implement several unit tests in order to validate the expected behavior of the implemented functions. More specifically, we test the computation of the density and velocity, the streaming function, mass preservation (i.e. first and second mass conservation equation and first and second impulse conservation equation from the Navier-Stokes Equations as well as a long run mass conservation test over 10000 time steps) and the boundary conditions. Additionally, we validate the parallelized implementation by comparing it to the serial implementation of the von Kármán's vortex street over 400 time steps using different number of nodes.

We integrate the unit tests into Continuous Integration (CI) using *Travis CI* as build server so that the implementation and its potential unintentional modifications are validated for each commit to the repository.

[1]`https://www.python.org/dev/peps/pep-0484/`

4

Numerical results

To demonstrate the LBM implementation we conduct several experiments with different combinations of boundary conditions. First we consider a *shear wave decay* (section 4.1) to validate whether our implementation preserves mass as well as to show how ω relates to the kinematic viscosity ν. In sections 4.2 and 4.3 we implement well known laminar flows, i.e. *couette* and *pouiseuille* flow, from the literature and compare it to their analytical solutions, respectively. In section 4.4 we implement the *von Kármán's vortex street* and in the following section 4.5 how we can reduce computational complexity by spatial domain decomposition as introduced in section 3.3.

4.1 Shear wave decay

The shear wave decay simulates a physical domain with only periodic boundary conditions which is in a initial state and its incremental steps towards the equilibrium state. One common practical and illustrative application of it, is the simulation of the breaking of a water dam. This exemplar application also shows the importance of simulations as the LBM to simulate rather than applying it in a real-world setting potentially causing mass destruction and high costs.

We choose the following simulation parameters for our experiments:

- lattice grid shape = 50×50
- $\omega = 1.0$
- Sinusoidal density in x-direction $\rho(\mathbf{x}, 0) = \rho_0 + \epsilon_\rho \sin(\frac{2\pi x}{l_x})$
  - $\rho_0(\mathbf{x}) = 0.5$
  - $\epsilon_\rho = 0.08$
  - $\mathbf{u}_{initial}(\mathbf{x}) = 0.0$
- Sinusoidal velocity in y-direction $\mathbf{u}_x(\mathbf{x}, 0) = \epsilon_\mathbf{u} \sin(\frac{2\pi y}{l_y})$

  - $\rho_{initial}(\mathbf{x}) = 0.0$
  - $\epsilon_{\mathbf{u}} = 0.08$

- time steps = 2000

4.2 Planar couette flow

The planar couette flow is a steady, laminar flow between two infinitely long, parallel plates with a fixed distance. One of those plates moves tangentially at a velocity of U relative to the other plate, which itself is stationary. The flow is caused by the viscous drag force acting on the fluid.

We choose the following simulation parameters for our experiments about the planar couette flow:

- lattice grid shape = 20×30
- $\omega = 1.0$
- $U = 0.05$
- $\rho_{initial}(\mathbf{x}) = 1.0$
- $\mathbf{u}_{initial}(\mathbf{x}) = 0.0$
- time steps = 4000

We can see that the simulated solution almost exactly reproduces the analytical solution. This shows that the bounce-back boundary condition is first-order accurate. Furthermore, it is also second-order accurate, since the boundary values and its first-order derivative are reproduced exactly.

We can also observe that the solution is viscosity-independent.

4.3 Planar poiseuille flow

The planar poiseuille flow is a steady flow between two non-moving plates. The flow is caused by a constant pressure gradient $\frac{dp}{dx}$ in the axial direction, x, parallel to two infinitely long parallel plates, seperated by a distance h.

We choose the following simulation parameters for our experiments about the planar poiseuille flow:

- lattice grid shape = 200×30
- $\omega = 1.5$
- $p_{out} = \frac{1}{3}$
- $\Delta p = 0.001111$
- $\rho_{initial}(\mathbf{x}) = 1.0$

- $\mathbf{u}_{initial}(\mathbf{x}) = 0.0$
- time steps = 5000

Some error can occur due to:

- Inaccuracy introduced by the bounce-back boundary condition. In particular, the error depends on the viscosity.[10]

In fig. xyz we can observe that the boundary velocity changes for different $\omega's$. This in turn shows that the poiseuille flow simulation is viscosity-dependent. This is called *numerical boundary slip.* [10]

4.4 Von Kármán's vortex street

4.5 Scaling tests

5

Conclusions

In this report we described the LBM and its implementation in Python. We also show several applications of the LBM to specific problems and compare the simulation results to analytical solutions if possible. We extend the serial implementation by using spatial domain decomposition to reduce computational costs by means of parallelization.

In chapter 2 we describe the theoretical foundations of the LBM. Starting from the continuous BTE we show how to discretize the BTE to obtain the LBM.

In chapter 3 we discussed the implementation and more specifically the parallelization of the LBM. We demonstrate by using the high-level language Python that implementations of the discrete equations are straightforward. We show how to reduce the computational burden by decompose the domain spatially into subdomains. Note, that the collision step is embarrassingly parallel. For the streaming step we use ghost cells to communicate adjacent lattice nodes to neighboring process. For the boundary conditions we transform global indices into local indicies in order to apply boundary conditions on the correct boundary nodes.

In chapter 4 we demonstrate in several applications the correctness of our LBM implementation by comparing the simulation results to analytical solutions from the literature. In addition, we demonstrate the speed-up that we can obtain from the spatial domain decomposition parallelization strategy of the LBM for the von Kármán's vortex street.

Bibliography

[1] McNamara and Zanetti. Use of the boltzmann equation to simulate lattice gas automata. *Physical review letters*, 61(20):2332–2335, 1988.

[2] Travis E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[3] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[4] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[5] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[6] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[7] Kerson Huang. *Statistical mechanics.* Wiley, New York, 2nd ed. edition, 1987.

[8] 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. *Physical Review*, 94(3):511–525, 1954.

[9] Benoît Paul Émile CLAPEYRON. *Mémoire sur la puissance motrice de la chaleur.* Journal de l'École Polytechnique, 1834.

[10] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The lattice Boltzmann method: Principles and practice / Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, Erlend Magnus Viggen.* Graduate texts in physics, 1868-4513. Springer, Switzerland, 2016.

[11] Sauro Succi. *The Lattice Boltzman Equation: For Complex States of Flowing Matter*, volume 1. Oxford University Press, 2018.

[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0. Contributors. Scipy 1.0–fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.

[13] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[14] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.