Implementation of a parallel solver of the Lattice Boltzman equation

Tidiane Camaret Ndir[1]

[1]Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Germany

1 Introduction

The Lattice Boltzmann method (LBM) is an efficient computational approach for fluid dynamics simulations. By discretizing the Boltzmann equation on a lattice grid, LBM enables modeling of complex fluid behaviors. This numerical method has proven highly effective for a wide range of applications.

However, LBM simulations can be computationally demanding, especially for high resolution modeling over large domains. To enable larger and more detailed simulations, implementing parallel computing techniques is essential to improve performance.

In this work, we develop and evaluate a parallel implementation of an LBM solver using domain decomposition. The simulation domain is divided into sub-domains and distributed across concurrent processes. While introducing parallelism complexity, this approach allows faster overall execution across distributed memory resources.

We first present the LBM methodology and our planned parallelization strategy. We then detail the parallel code implementation using MPI libraries for inter-process coordination. We provide an analysis of the speedup and scalability of our approach. The code and running instructions are available at `https://github.com/tidiane-camaret/hpc_fluid_dynamics`.

2 Methodology

2.1 General principles

The Lattice Boltzmann method (LBM) is based on the discretized Boltzmann transport equation for fluid dynamics. LBM represents the fluid as pseudo-particles that propagate and collide on a discrete lattice grid. Instead of modeling the trajectories of every particle of the domain, we model the probability density function over the phase space, which is the set of all possible positions (a grid of size NX, NY) and velocities (9 directions that point to the adjacent positions, written as c_i, and their respective weights w_i.

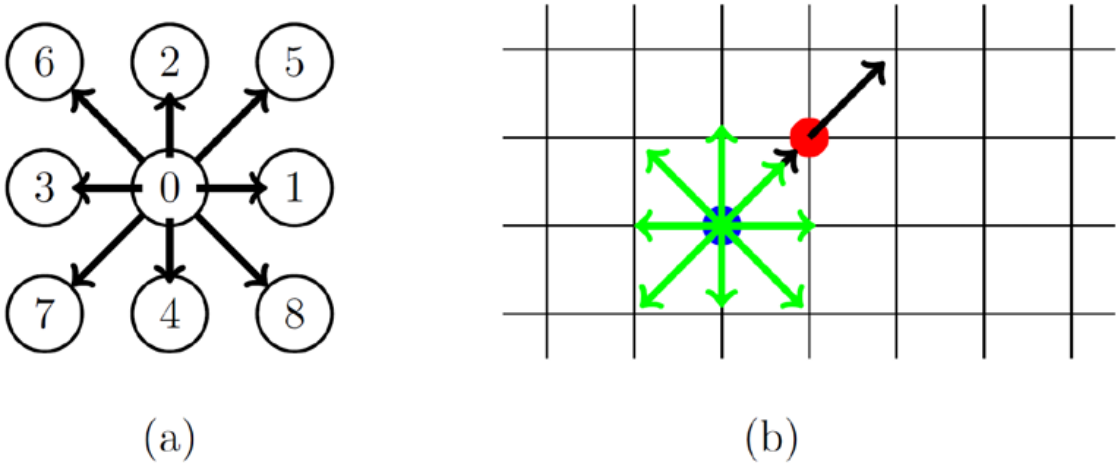

Figure 1: (a) Discretization on the velocity space according to D2Q9. (b) Uniform 2D grid for the discretization in the physical space. Source : High-Performance Computing: Fluid Mechanics with Python, A.Greiner

f is therefore of size $9 * NX * NY$. At a time step t, it fully describes the state of the lattice. The LBM evolves the particle distribution function iteratively, through a series of alternating streaming and collision steps. In the streaming step, f propagates to neighboring lattice sites based on particle velocity directions:

$$f_i(x + c_i dt, t + dt) = f_i(x, t)$$

where f_i is the distribution for velocity c_i. This moves particles to the next grid points along their travel direction. In the collision step, f relaxes towards an equilibrium distribution f_{eq} via the Bhatnagar–Gross–Krook (BGK) approximation:

$$f_i(x, t + dt) = f_i(x, t) - \omega(f_i - f_{eq})$$

where ω is a relaxation parameter and f_ρ is calculated from fluid density and velocity. Collisions cause f to locally relax towards equilibrium. The equilibrium distribution f_{eq} is typically modeled as:

$$f_{eq} = \rho w_i [1 + 3(c_i u)/c^2 + 9(c_i u)^2/2c^4 - 3u^2/2c^2]$$

where ρ is the fluid density, u its velocity, and c the particle speed. Alternating these streaming and collision steps simulates the evolution of the fluid system.

2.2 The parallelization strategy

In order to parallelize the LBM computations, we utilize a spatial domain decomposition approach. The simulation domain is divided into equal sub-domains along the X- and Y-dimensions. Each sub-domain is assigned to a MPI process running on separate CPU cores, identified by its rank number. Buffer regions allow communication between adjacent sub-domains to synchronize boundary values. 2D decomposition provides a straightforward strategy to distribute the workload with localized data access.

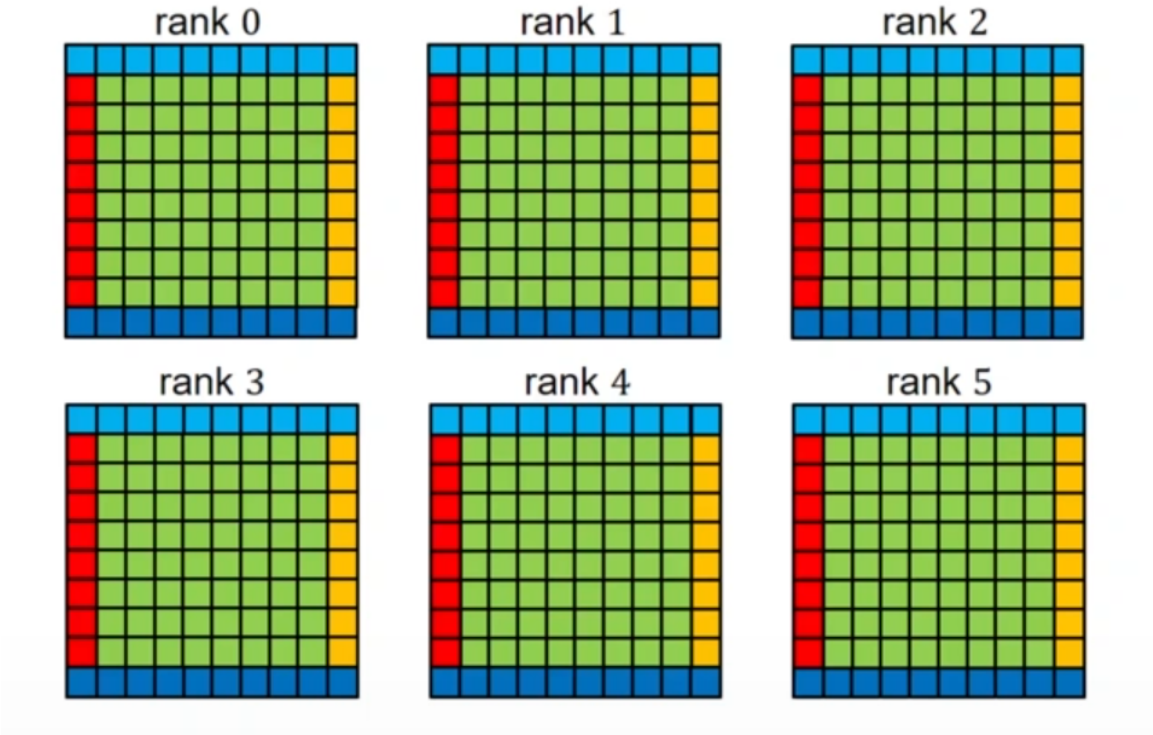

Figure 2: Example of a domain subdivision. The edge regions are buffer zones. Source : High-Performance Computing: Fluid Mechanics with Python, A.Greiner

3 Implementation

3.1 Initialisation and object definition

The code is structured around the LBM class defined in *hpc_fluid_dynamics/lbm_class.py*. when instantiated, it initializes the lattice according to the following parameters :

- NX, NY : dimensions of the grid
- ω: the relaxation parameter
- c_i : the list of grid velocity vectors = $[(0,0),(1,0),(0,1),(-1,0),(0,-1),(1,1),(-1,1),(-1,-1),(1,-1)]$
- w_i : the weight associated to each velocity vector = $[4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36]$

The array pdf_{9xy}, representing the density f, is initialized with dimensions $(9, NX, NY)$. We can then run the simulation for a defined number of steps, using the run() method.

```
def __init__(self,
                 NX = 250, # number of lattice points in the x direction
                 NY = 250, # number of lattice points in the y direction
                 mode = "circle", # initial condition mode (see lbm_utils.py)
                 omega = 0.1, # relaxation parameter
                 parallel = False, # parallel simulation
                 epsilon = 0.05, # perturbation amplitude
             ):

        """
        Initialize the parameters of the LBM.
        """
        self.NX = NX
        self.NY = NY
        self.mode = mode
        self.parallel = parallel
        self.omega = omega
        self.epsilon = epsilon
        self.viscosity = 1/3*(1/omega - 0.5)

        # set the constants
        self.velocity_set = np.array([[0, 0], [1, 0], [0, 1], [-1, 0], [0, -1],
                                      [1, 1], [-1, 1], [-1, -1], [1, -1]])
        self.velocity_set_weights = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36,
                                              1/36, 1/36, 1/36])
        self.sound_speed = 1 / np.sqrt(3)
        self.wall_velocity = np.array([0.1, 0])
        self.p_in = 0.1
        self.p_out = 0.01
        self.d_p = self.p_out - self.p_in
        self.density_in = (self.p_out + self.d_p) / sound_speed**2
        self.density_out = (self.p_out) / sound_speed**2

        # Initialize the PDF
        self.pdf_9xy = init_pdf(NX, NY, mode, self.epsilon)

        # Flags for boundary conditions on each edge
        self.is_boundary = {
            "left": True,
            "right": True,
            "top": True,
            "bottom": True,
```

```
        }
```

Listing 1: Initialization and object definition

3.2 Initialization of the Message passing interface

We create an MPI communicator using *MPI.COMM_WORLD*, and define A 2D Cartesian topology to map processes to a grid. Ranks are assigned coordinates and neighbors.

```
        # Initialize the MPI communicator
        if parallel:
            self.comm = MPI.COMM_WORLD
            self.rank = self.comm.Get_rank()
            self.size = self.comm.Get_size()

            ### Domain decomposition
            if NX < NY:
                sectsX=int(np.floor(np.sqrt(self.size*NX/NY)))
                sectsY=int(np.floor(self.size/sectsX))
                print('We have {} fields in x-direction and {} in y-direction'.
    format(sectsX,sectsY))
                print('How do the fractions look like?')
                print('NX/NY={} and sectsX/sectsY = {}\n'.format(NX/NY,sectsX/
    sectsY))
            elif NX > NY:
                sectsY=int(np.floor(np.sqrt(self.size*NY/NX)))
                sectsX=int(np.floor(self.size/sectsY))
                print('We have {} fields in x-direction and {} in y-direction'.
    format(sectsX,sectsY))
                print('How do the fractions look like?')
                print('NX/NY={} and sectsX/sectsY = {}\n'.format(NX/NY,sectsX/
    sectsY))
            elif NX==NY:
                sectsY=int(np.floor(np.sqrt(self.size)))
                sectsX=int(self.size/sectsY)
                if self.rank == 0: print('In the case of equal size we divide the
    processes as {} and {}'.format(sectsX,sectsY))

            self.sectsX=sectsX
            self.sectsY=sectsY
            self.nxsub = NX//self.sectsX+2
            self.nysub = NY//self.sectsY+2
            self.cartcomm=self.comm.Create_cart(dims=[sectsX,sectsY],periods=[True,
    True],reorder=False)
            self.rcoords = self.cartcomm.Get_coords(self.rank)

            # Flags for boundary conditions on each edge
            self.is_boundary = {
                'left': self.rcoords[0] == 0,
                'right': self.rcoords[0] == self.sectsY - 1,
                'top': self.rcoords[1] == self.sectsX - 1,
                'bottom': self.rcoords[1] == 0,
            }

            # where to receive from and where send to
            sR,dR = self.cartcomm.Shift(1,1)
            sL,dL = self.cartcomm.Shift(1,-1)

            sU,dU = self.cartcomm.Shift(0,-1)
```

```
            sD,dD = self.cartcomm.Shift(0,1)

            self.sd = np.array([sR,dR,sL,dL,sU,dU,sD,dD], dtype = int)

            allrcoords = self.comm.gather(self.rcoords,root = 0)
            allDestSourBuf = np.zeros(self.size*8, dtype = int)
            self.comm.Gather(self.sd, allDestSourBuf, root = 0)

            if self.rank == 0:

                self.density_plot_list = []
                print(allrcoords)
                print(' ')
                cartarray = np.ones((sectsY,sectsX),dtype=int)
                allDestSour = np.array(allDestSourBuf).reshape((self.size,8))
                for i in np.arange(self.size):
                    cartarray[allrcoords[i][0],allrcoords[i][1]] = i
                    print('Rank {} all destinations and sources {}'.format(i,
    allDestSour[i,:]))
                    sR,dR,sL,dL,sU,dU,sD,dD = allDestSour[i]
                    print('Rank {} is at {}'.format(i,allrcoords[i]))
                    print('sour/dest right {} {}'.format(sR,dR))
                    print('sour/dest left  {} {}'.format(sL,dL))
                    print('sour/dest up    {} {}'.format(sU,dU))
                    print('sour/dest down  {} {}'.format(sD,dD))
                print('')
                print(cartarray)

            # separate the pdf into subdomains
            self.pdf_9xy = self.pdf_9xy[:,self.rcoords[0]*NY//sectsY:(self.rcoords
    [0]+1)*NY//sectsY,self.rcoords[1]*NX//sectsX:(self.rcoords[1]+1)*NX//sectsX]
```

Listing 2: Initialization of the Message passing interface

3.3 Communication between Processes

PDF sub-domains exchange information using the *Sendrecv* command.

```
def Communicate(pdf_9xy,cartcomm,sd,is_boundary):
    recvbuf = np.zeros(pdf_9xy[:,:,1].shape)
    sR,dR,sL,dL,sU,dU,sD,dD = sd
    # Send to right which is destination rigth (dR) and receive from left which is
    source right (sR)
    # print(rank,'Right, source',sR,'destination',dR)
    sendbuf = pdf_9xy[:,:,-2].copy() # Send the second last column to dR
    cartcomm.Sendrecv(sendbuf, dR, recvbuf = recvbuf, source = sR)
    if not is_boundary["bottom"]:
        pdf_9xy[:,:,0] = recvbuf # received into the 0th column from sR
    # Send to left and receive from right
    #print(rank,'Left, source',sL,'destination',dL)
    sendbuf = pdf_9xy[:,:,1].copy()
    cartcomm.Sendrecv(sendbuf, dL, recvbuf = recvbuf, source = sL)
    if not is_boundary["top"]:
        pdf_9xy[:,:,-1] = recvbuf
    # Send to up and receive from down
    #print(rank,'Up, source',sU,'destination',dU)
    sendbuf = pdf_9xy[:,1,:].copy()
    cartcomm.Sendrecv(sendbuf, dU, recvbuf = recvbuf, source = sU)
    if not is_boundary["right"]:
        pdf_9xy[:,-1,:] = recvbuf
```

```
    # Send to down and receive from up
    #print(rank,'Down, source',sD,'destination',dD)
    sendbuf = pdf_9xy[:,-2,:].copy()
    cartcomm.Sendrecv(sendbuf, dD, recvbuf = recvbuf, source = sD)
    if not is_boundary["left"]:
        pdf_9xy[:,0,:]=recvbuf
#
    return pdf_9xy
```

Listing 3: Communication between Processes

3.4 Collision and streaming steps

Local density and velocity are computed from PDFs via summation. Equilibrium distributions are calculated, and used in the collide step. The stream step shifts PDFs between lattice sites.

```
for i in range(self.nt):
        [...]
    # COMMUNICATE
    if self.parallel:
        pdf_9xy = Communicate(pdf_9xy,self.cartcomm,self.sd,self.is_boundary)

    # MOMENT UPDATE
    density_xy = calc_density(pdf_9xy)
    local_avg_velocity_xy2 = calc_local_avg_velocity(pdf_9xy,density_xy)

    # EQULIBRIUM
    equilibrium_pdf_9xy = calc_equilibrium_pdf(density_xy, local_avg_velocity_xy2)

    [...]

    # COLLISION STEP
    pdf_9xy = pdf_9xy + self.omega*(equilibrium_pdf_9xy - pdf_9xy)

    # STREAMING STEP
    pdf_9xy = streaming(pdf_9xy)

    # BOUNDARY CONDITIONS
    pdf_9xy = self.boundary_conditions(pdf_9xy, density_xy, local_avg_velocity_xy2,
     equilibrium_pdf_9xy)
```

Listing 4: Collision and streaming steps

3.5 Boundary conditions

Processes at global edges identify their boundary roles. Bounce-back is implemented by swapping PDFs. Velocity conditions adjust inflow/outflow based on macro values. Exchanges propagate updated boundaries.

```
    def boundary_conditions(self,pdf_9xy,density_xy):
        [...]
        if self.mode == 'lid':
            opposite_indexes = [[5, 7], [1, 3], [8, 6]] # indexes of opposite
    directions
            # bounce back conditions on the left wall
            if self.is_boundary["left"]:
                for oi in opposite_indexes:
                    pdf_9xy[oi[0], 0, :] = pdf_9xy[oi[1], 0, :]
            # bounce back conditions on the right wall
```

```
            if self.is_boundary["right"]:
                for oi in opposite_indexes:
                    pdf_9xy[oi[1], -1, :] = pdf_9xy[oi[0], -1, :]

        if self.mode == "poiseuille":
            opposite_indexes = [[6, 8], [2, 4], [5, 7]] # indexes of opposite directions
            # bounce back conditions on the lower wall
            if self.is_boundary["bottom"]:
                for oi in opposite_indexes:
                    pdf_9xy[oi[0], :, 0] = pdf_9xy[oi[1], :, 0]

            # bounce back conditions on the upper wall
            if self.is_boundary["top"]:
                for oi in opposite_indexes:
                    pdf_9xy[oi[1], :, -1] = pdf_9xy[oi[0], :, -1]
```

Listing 5: Boundary conditions

3.6 Gathering and saving results

The local density and velocity data from each subdomain is flattened into 1D arrays.

MPI_Gather collects the subsets from all processes onto the root rank, storing them in full 1D arrays.

Process grid coordinates are gathered to root as well.

On root, loops reconstitute the full 2D data by inserting each sub-domain's contribution based on its coordinates.

This reconstructs the complete parallel results on root for output and analysis.

Non-root processes simply save their local subsets.

Gathering with coordinates enables rebuilding the global solution from the distributed process-local data after domain decomposition.

```
if self.parallel:
    # GATHER AND SAVE RESULTS
    density_1D = np.zeros((self.NX*self.NY)) # 1D array to store density
    velocity_1D = np.zeros((self.NX*self.NY,2)) # 1D array to store velocity
    self.comm.Gather(density_xy.reshape((self.nxsub-2)*(self.nysub-2)), density_1D, root = 0)
    self.comm.Gather(local_avg_velocity_xy2.reshape((self.nxsub-2)*(self.nysub-2),2), velocity_1D, root = 0)

    rcoords_x = self.comm.gather(self.rcoords[1], root=0)
    rcoords_y = self.comm.gather(self.rcoords[0], root=0)
    if self.rank == 0:

        xy = np.array([rcoords_x,rcoords_y]).T
        density_xy_gathered = np.zeros((self.NX,self.NY))
        velocity_xy_gathered = np.zeros((self.NX,self.NY,2))

        for i in np.arange(self.sectsX):
            for j in np.arange(self.sectsY):
                k = i*self.sectsX+j
                xlo = self.NX//self.sectsX*xy[k,1]
                xhi = self.NX//self.sectsX*(xy[k,1]+1)
                ylo = self.NY//self.sectsY*xy[k,0]
                yhi = self.NY//self.sectsY*(xy[k,0]+1)
                clo = k*self.NX*self.NY//(self.sectsX*self.sectsY)
                chi = (k+1)*self.NX*self.NY//(self.sectsX*self.sectsY)
```

```
25
26                  density_xy_gathered[xlo:xhi,ylo:yhi] = density_1D[clo:chi].reshape(
    self.NX//self.sectsX,self.NY//self.sectsY)
27                  velocity_xy_gathered[xlo:xhi,ylo:yhi,:] = velocity_1D[clo:chi,:].
    reshape(self.NX//self.sectsX,self.NY//self.sectsY,2)
28
29          self.densities.append(density_xy_gathered)
30          self.velocities.append(velocity_xy_gathered)
31
32
33  else:
34      # SAVE RESULTS
35      self.densities.append(density_xy)
36      self.velocities.append(local_avg_velocity_xy2)
```

Listing 6: Gathering and saving results

4 Results

We are able to run the simulation under various parameters and initial conditions :

4.1 Shear wave decay

This method consists in setting a sinusoidal velocity profile in our lattice, and measuring how fast it decays. The initial conditions for the density and velocity are as follows : $\rho(r, 0) = 1$ and $u_x(r, 0) = \epsilon(\frac{2\pi y}{L_y})$, with $\epsilon = 0.05$. We run the simulation using the following command :

```
python3 scripts/run_lbm.py --mode shear_wave --nt 3000
```

Listing 7: bash command for the shear wave decay

This script generates gif animations for velocity and density, available in the **results** directory, as well as further analysis of the profiles of the distribution :

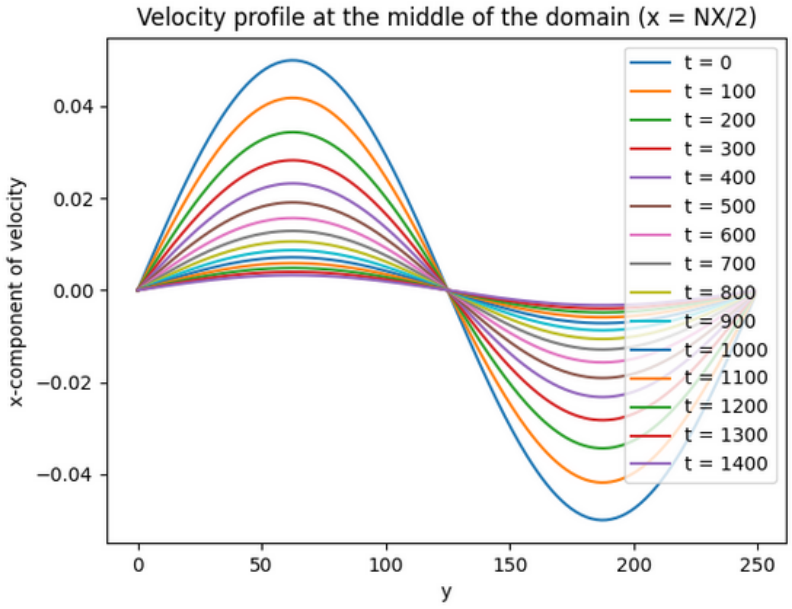

Figure 3: Shear wave : x component of the shear wave velocity profile (x = NX//2). It decreases over time.

The x component of the velocity profile (x = NX//2) decreases over time, while the y component of the velocity profile stays at zero over time. The density profile does not vary over time, which can be expected since velocity stays uniform across the x dimension.

We can also measure the amplitude of the velocity over time : $max(|v||) - min(||v||)$, and calculate the empirical viscosity via the relation $amplitude = \epsilon * \exp(-viscosity * (\frac{2*\pi}{NX})^2 t)$ We can compare it to the theoretical viscosity that depends on ω via the formula $\nu = \frac{1}{3} * (\frac{1}{\omega} - \frac{1}{2})$. Figure 6 shows the measure and theoretical viscosity for different values of omega.

We run the simulation using the following command :

```
python3 scripts/run_viscosity_comparison.py
```

Listing 8: bash command for the viscosity comparison

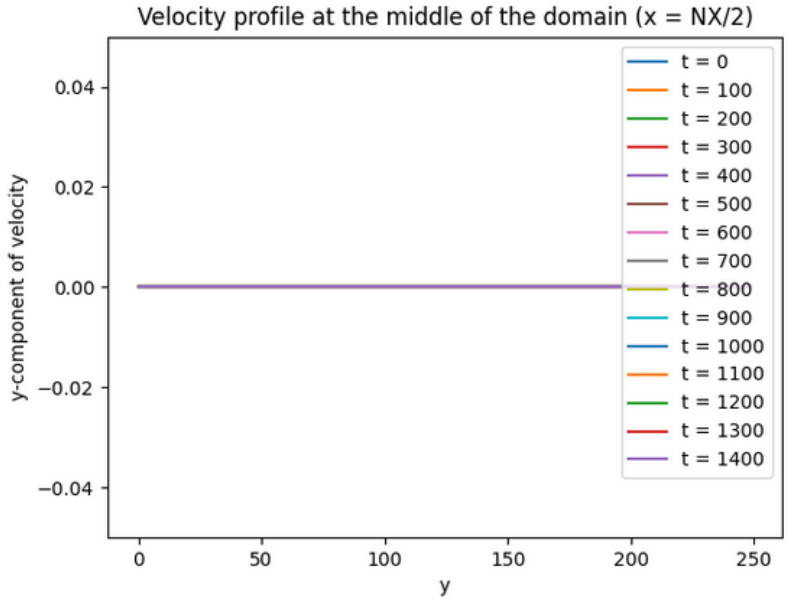

Figure 4: Shear wave : y component of the shear wave velocity profile. It does not vary over time.

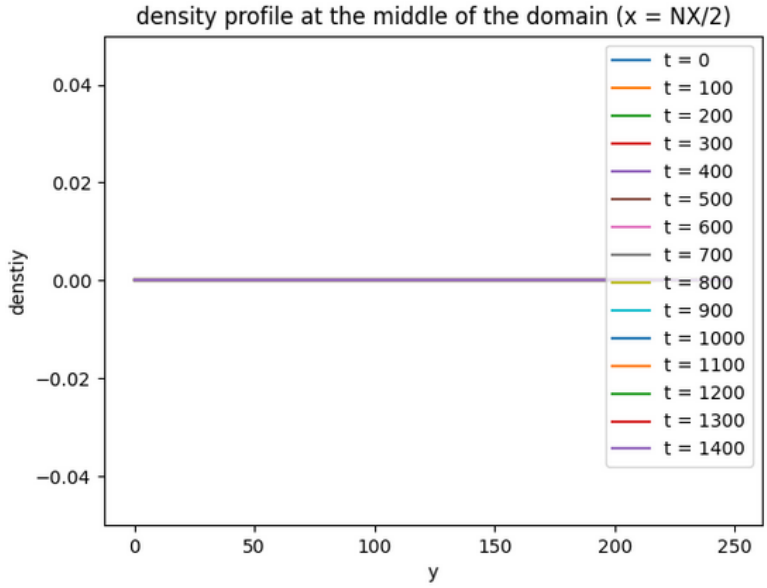

Figure 5: Shear wave density profile. It does not vary over time.

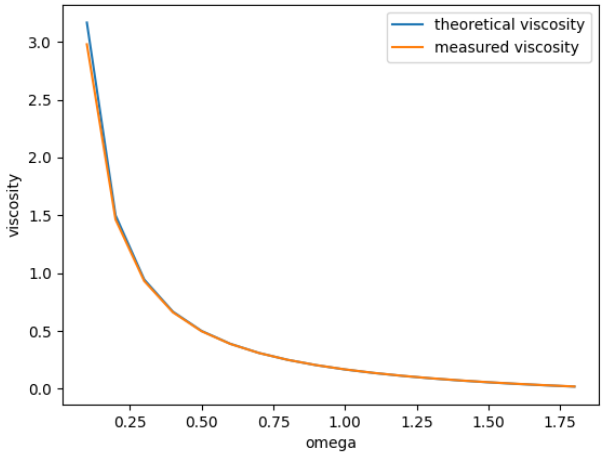

Figure 6: Shear wave : Measured and analytic viscosity for different values of ω

4.2 Couette flow

In this scenario, the fluid flows between a fixed and a moving wall, set here at a speed of $[0.1, 0]$. The initial conditions for the density and velocity are $\rho(r, 0) = 1$ and $u_x(r, 0) = 0$, and we apply the following conditions for the boundaries :

- bounce-back boundary conditions on the lower wall (**on the left** in the animation graphs)
- bounce-back and moving boundary conditions on the upper wall (**on the right** in the animation graphs).
- periodic boundary conditions on left and right walls (**on the top and bottom** in the animation graphs).

We run the simulation using the following command :

```
python3 scripts/run_lbm.py --mode couette --nt 3000
```

Listing 9: bash command for the couette flow

Figure 7 shows the evolution of the velocity profile over time.

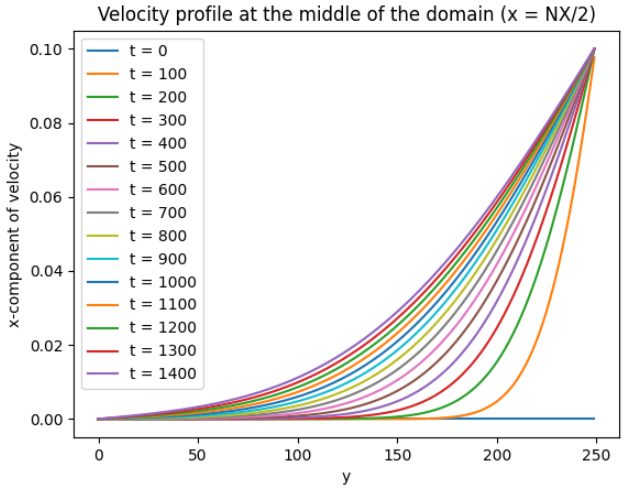

Figure 7: x component of the Couette flow velocity profile (x = NX//2). It increases over time.

4.3 Poiseuille flow

In this scenario, the fluid flows between the two fixed walls of a tunnel, and a pressure difference is applied between each end. The initial conditions for the density and velocity are $\rho(r, 0) = 1$ and $u_x(r, 0) = 0$, and we apply the following conditions for the boundaries :

- bounce-back boundary conditions on the lower wall (**on the left** in the animation graphs)
- bounce-back boundary conditions on the upper wall (**on the right** in the animation graphs).
- periodic boundary conditions with pressure variation on left and right walls (**on the top and bottom** in the animation graphs).

We run the simulation using the following command :

```
python3 scripts/run_lbm.py --mode poiseuille --NX 200 --NY 50 --nt 1000 --omega 1.2
```

Listing 10: bash command for the Poiseuille flow

Figure 8 shows the evolution of the velocity profile over time, as well as the analytic calculation of the profile

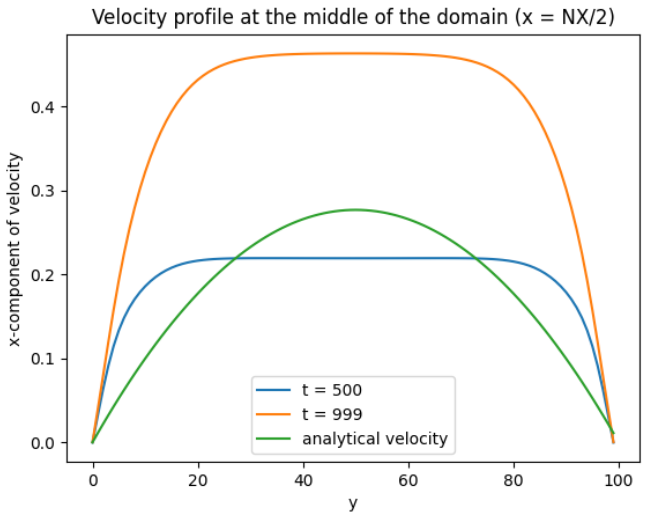

Figure 8: x component of the Poiseuille flow velocity profile (x = NX//2). It increases over time.

4.4 Sliding Lid

In this scenario, the fluid is contained in a box which lid is given a certain velocity. The initial conditions for the density and velocity are $\rho(r, 0) = 1$ and $u_x(r, 0) = 0$, and we apply the following conditions for the boundaries :

- bounce-back, moving boundary conditions on the upper wall (**on the right** in the animation graphs).
- periodic boundary conditions with pressure variation on left, right and lower walls (**on the top, bottom and left** in the animation graphs).

We run the simulation using the following command :

```
python3 scripts/run_lbm.py --mode lid --NX 200 --NY 200 --nt 1000 --omega 0.02
```

Listing 11: bash command for the Sliding lid

Figure 9 shows the streams of lid scenario.

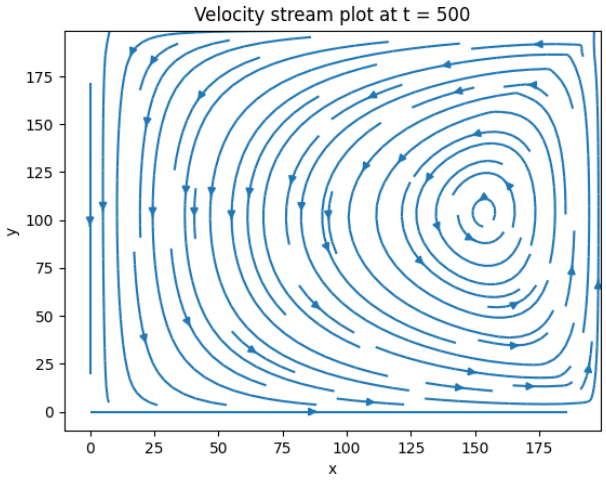

Figure 9: Streaming plot of the Sliding lid scenario

4.5 Parallelization

We can run all previous experiments on multiple processes using the MPI, by adding the **–parallel** flag to the scripts. For example :

```
mpirun -np 16 python3 scripts/run_lbm.py --mode lid --NX 200 --NY 200 --nt 1000 --
    omega 0.02 --parallel
```

Listing 12: bash command for the Sliding lid, in parallel. The –np flag sets the number of processes.

We can also measure the speedup provided by the multiprocessing, using the following script :

```
time python3 scripts/time_measure.py
```

Listing 13: bash command for the timing benchmark

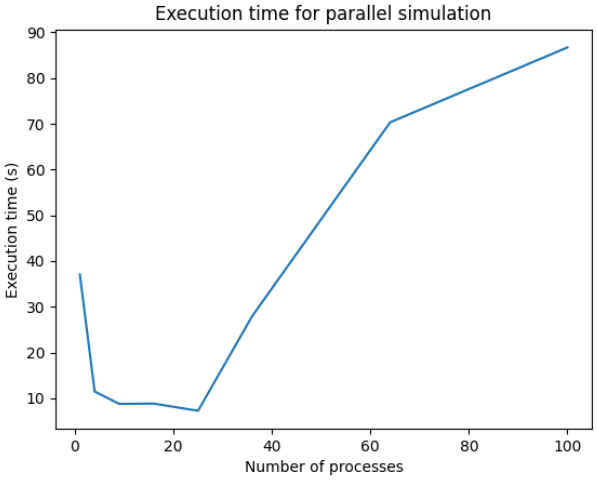

Figure 10: Streaming plot of the Sliding lid scenario

Figure 10 shows the execution times for various number of processes. We observe that up until 25 processes, adding processes speeds up the simulation. Over this number, the amount of information to pass between processes actually slows down the simulation.

5 Summary

This work implemented a parallel Lattice Boltzmann Method (LBM) fluid flow simulator using a spatial domain decomposition approach with MPI. We divided the computational domain into sub-domains mapped to a 2D Cartesian grid of processes, evolving the streaming and collision steps locally, and exchanging buffer regions between sub-domains to synchronize boundaries.

We were able to significantly decrease the execution time by using the MPI approach. However, it gives an advantage up until a certain number of parallel processes only. It is therefore necessary to run tests to calibrate this parameter correctly.

References

[Krü+17] Timm Krüger et al. *The Lattice Boltzmann Method: Principles and Practice*. en. Graduate Texts in Physics. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-44647-9 978-3-319-44649-3. DOI: 10.1007/978-3-319-44649-3. URL: http://link.springer.com/10.1007/978-3-319-44649-3 (visited on 08/14/2023).

[FLL18] Linlin Fei, Kai H. Luo, and Qing Li. "Three-dimensional cascaded lattice Boltzmann method: Improved implementation and consistent forcing scheme". In: *Physical Review E* 97.5 (May 2018). Publisher: American Physical Society, p. 053309. DOI: 10.1103/PhysRevE.97.053309. URL: https://link.aps.org/doi/10.1103/PhysRevE.97.053309.

[PG] Lars Pastewka and Andreas Greiner. "HPC with Python: An MPI-parallel implementation of the Lattice Boltzmann Method". en. In: ().

[] *The lattice Boltzmann Equation - Sauro Succi - Librairie Eyrolles*. URL: https://www.eyrolles.com/Sciences/Livre/the-lattice-boltzmann-equation-9780198503989/.