

HPC with Python: An MPI-parallel implementation of the Lattice Boltzmann Method

Lars Pastewka 

Andreas Greiner

Department of Microsystems Engineering, University of Freiburg, Germany

The Lattice Boltzmann Method is well suited for high-performance computational fluid dynamics. We show by means of a common two-dimensional test case, the lid-driven cavity problem, that excellent parallel scaling can be achieved in an implementation based on pure Python, using the `numpy` library and the Message Passing Interface. We highlight opportunities and pitfalls for the implementation of parallel high-performance codes in the high-level language Python.

1 Introduction

The Boltzmann transport equation (BTE) was introduced by Ludwig Boltzmann in the context of kinetic gas theory (Boltzmann, 1896) and is a statistical model for the transport of molecular constituents during flow (Cercignani, 1988). The *Lattice* Boltzmann method (LBM) is a numerical scheme based on a discretized version of the BTE, introduced by McNamara and Zanetti in 1988 (McNamara et al., 1988). LBM models have been used for the last three decades to study the dynamics of fluids in many different applications, from multiphase flow (Gunstensen, Rothman et al., 1991; Grunau et al., 1993), porous media (Aharonov et al., 1993; Gunstensen and Rothman, 1993) to microfluidics (Zhang, 2011). A thorough review of its applications in fluid dynamics and beyond can be found in Succi (2001).

2 The Boltzmann transport equation

The BTE describes the time of evolution of the probability density $f(\mathbf{v}, \mathbf{r}, t)$ for finding a molecule with mass m and velocity \mathbf{v} at position \mathbf{r} as a function of time t . The moments of this probability density define the mass density $\rho(\mathbf{r}, t)$, the momentum density $\mathbf{j}(\mathbf{r}, t)$ and the temperature $T(\mathbf{r}, t)$ in D -dimensional space,

$$\rho(\mathbf{r}, t) = m \int d^D v f(\mathbf{v}, \mathbf{r}, t) \quad (1)$$

$$\mathbf{j}(\mathbf{r}, t) = \rho(\mathbf{r}, t) \mathbf{u}(\mathbf{r}, t) = m \int d^D v \mathbf{v} f(\mathbf{v}, \mathbf{r}, t) \quad (2)$$

$$k_B T(\mathbf{r}, t) = \frac{m^2}{D \rho(\mathbf{r}, t)} \int d^D v [\mathbf{v} - \mathbf{u}(\mathbf{r}, t)]^2 f(\mathbf{v}, \mathbf{r}, t), \quad (3)$$

where we have defined the average velocity $\mathbf{u}(\mathbf{r}, t)$ at position \mathbf{r} . We have here assumed a monoatomic system with molecules of mass m and k_B is the Boltzmann constant.

The BTE describes the total time-rate of change of this probability distribution, df/dt . We know that for large times t it must relax towards statistical equilibrium, given by the Maxwell velocity distribution function (Huang, 1987),

$$f^{\text{eq}}(\mathbf{v}; \rho, \mathbf{u}, T) = \frac{\rho}{m} \left(\frac{m}{2\pi k_B T} \right)^{D/2} \exp \left\{ -\frac{m(\mathbf{v} - \mathbf{u})^2}{2k_B T} \right\}. \quad (4)$$

A common approximation is to assume relaxation of f towards f^{eq} with a single characteristic time τ , as suggested by Bhatnagar, Gross, and Krook (BGK) in 1954 (Bhatnagar et al., 1954),

$$\frac{df(\mathbf{v}, \mathbf{r}, t)}{dt} = -\frac{f(\mathbf{v}, \mathbf{r}, t) - f^{\text{eq}}(\mathbf{v}; \rho(\mathbf{r}, t), \mathbf{u}(\mathbf{r}, t), T(\mathbf{r}, t))}{\tau}. \quad (5)$$

Eq. (5) is the BGK-Boltzmann equation. Note that the total differential of f is

$$\frac{df}{dt} = \frac{\mathbf{F}(\mathbf{r})}{m} \frac{\partial f(\mathbf{v}, \mathbf{r}, t)}{\partial \mathbf{v}} + \mathbf{v} \frac{\partial f(\mathbf{v}, \mathbf{r}, t)}{\partial \mathbf{r}} + \frac{\partial f(\mathbf{v}, \mathbf{r}, t)}{\partial t}, \quad (6)$$

where $\mathbf{F}(\mathbf{r})$ is an external force acting on the molecules.

We will remain within the context of this single (BGK) relaxation time approximation, but modern developments of the method aim at introducing multiple relaxa-

tion times, for example by relaxing the cumulants of f individually (Geier, Greiner et al., 2006; Geier, Schönherr et al., 2015).

2.1 The Lattice Boltzmann Method in two dimensions

The BTE is discretized in space, velocity and time. Discretizing Eq. (5) on a regular (square) lattice in space is straightforward. However, we also require a suitable discretization of velocity space and the time step. Both are chosen such that the distance between interpolation points in velocity space multiplied by the time step equals the distance between points on the spatial lattice; in other words, a molecule traveling on the spatial lattice travels between integer number of lattice points during one time step.

The particular realization of the velocities used by us is shown in Fig. 1a. The velocity set contains 9 directions. Each direction is assigned the index shown in Fig. 1a. Direction 0 zero hence describes the population of molecules at rest. This specific discretization in two-dimensional space ($D = 2$) with nine directions is commonly denoted by D2Q9.

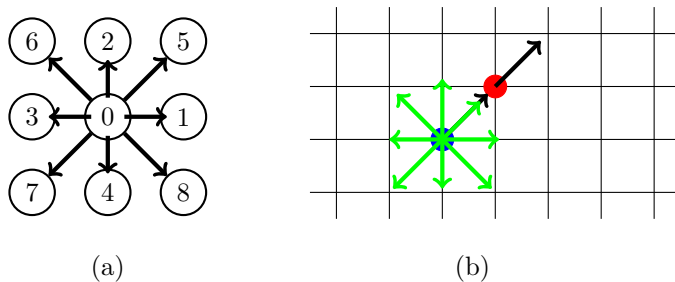


Figure 1: Discretization of the BTE. (a) Discretization of velocity space into nine directions. The numbers uniquely identify the direction. (b) Regular two-dimensional lattice used for the spatial discretization.

The discrete velocities are therefore given by the directions to the eight neighbors divided by the time step. We have nine velocity vectors \mathbf{c}_i , with $i = 0, \dots, 8$ for each direction i . We now also require the occupation numbers for these nine directions. The probability distribution $f(\mathbf{r}, \mathbf{v})$ is hence represented by nine discrete $f_i(\mathbf{x}_j, t)$

where \mathbf{x}_j is the discrete lattice point and i denotes the direction. Note that the moments of the probability density Eqs. (1) and (2) become

$$\rho(\mathbf{x}_j, t) = \sum_i f_i(\mathbf{x}_j, t) \quad (7)$$

$$\mathbf{u}(\mathbf{x}_j, t) = \frac{1}{\rho(\mathbf{x}_j, t)} \sum_i \mathbf{c}_i f_i(\mathbf{x}_j, t), \quad (8)$$

where $\rho(\mathbf{x}_j, t)$ is now the number density, i. e. we assume unit molecular mass.

The discretized BGK-Boltzmann equation (5) then reads

$$f_i(\mathbf{x}_j + \mathbf{c}_i \cdot \Delta t, t + \Delta t) = f_i(\mathbf{x}_j, t) - \omega [f_i(\mathbf{x}_j, t) - f_i^{\text{eq}}(\mathbf{x}_j, t)] \quad (9)$$

where Δt is a time step used for the discrete dynamical propagation of the occupation numbers. We have introduced the normalized relaxation parameter $\omega = \Delta t/\tau$ and the external forces $\mathbf{F}(\mathbf{r})$ are set to zero. Note that the left hand side and the first term on the right hand side of Eq. (9) is a discretization of the total differential of f . The expression for the equilibrium distribution function in the nine directions is (Mohamad, 2011; Wolf-Gladrow, 2000)

$$f_i^{\text{eq}}(\mathbf{x}_j, t) = w_i \rho(\mathbf{x}_j, t) \left\{ 1 + 3\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x}_j, t) + \frac{9}{2} [\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x}_j, t)]^2 - \frac{3}{2} u^2(\mathbf{x}_j, t) \right\}, \quad (10)$$

with $w_i = 4/9, 1/9$ and $1/36$ for directions 0, 1-3 and 4-8, respectively. Note that like Eq. (4) for the continuous distribution function, Eq. (10) is obtained by applying the maximum entropy principle under the constraint of mass and momentum conservation to the discrete distribution function (Mohamad, 2011; Wolf-Gladrow, 2000).

To give a rough idea of how Eq. (9) propagates the occupation numbers for the individual directions, we describe as an example what happens to f_5 . Assume we have a finite value for direction f_i including f_5 , their magnitude given by the length of the green arrows, respectively, and based at the position shown by the blue spot in Fig. 1b. At time t the r.h.s. of Eq. (9) relaxes f_5 towards the local equilibrium given by the black arrow based at the blue spot, which is commonly called the collision operation (Boltzmann, 1896). Time propagation is described by the l.h.s. of Eq. (9). After one time step Δt , the occupation f_5 will be handed to the red spot and occupy direction 5 there. This is commonly called the streaming step. An

algorithmic implementation of Eq. (9) is conveniently split into these streaming and collision steps.

2.2 Implementation in Python and numpy

The quantity that specifies the state of our simulation are the occupation numbers $f_i(\mathbf{x}_j)$. We represent these as a single contiguous `numpy`¹ array called `f_ikl`. Note that the suffixes in our specific naming scheme indicate the number of array dimensions and their function. In the present case, `ikl` stands for three array dimensions: `i` is a direction (array dimension of size 9), `k` is the lattice position in x -direction, and `l` (size n_x) the lattice position in y -direction (size n_y). Below we also encounter `c`, which denotes a Cartesian array dimension (size 2). This naming convention eases readability of the code and translation of formulas containing linear algebra into `numpy` operations and is borrowed from the GPAW code (Mortensen et al., 2005; Enkovaara et al., 2010). Note that because `numpy`'s default storage order is row-major, the array `f_ikl` is stored in what is commonly called the structure of arrays (SoA) storage order (Obrecht et al., 2011; Qi, 2017).

The streaming part can be cast into the form listed in Code Listing 1, which uses the `numpy.roll` function and automatically takes care of periodic boundary conditions. `numpy.roll` rolls the data on the lattice into the direction specified by `axis` in the code listing by a distance given by `c_ic[i]`. Note that `c_ic[i]` is a one-dimensional array of length 2 that specifies the Cartesian coordinates of direction i . Taking the example of the `f_ikl` array, a roll operation on occupation number 1 and `axis` 0 with unit distance will assign $f_{1,k,l} \rightarrow f_{1,k+1,l}$.

```
import numpy as np
c_ic = np.array([[0, 1, 0, -1, 0, 1, -1, -1, 1],
                [0, 0, 1, 0, -1, 1, 1, -1, -1]]).T

def stream(f_ikl):
    for i in range(1, 9):
        f_ikl[i] = np.roll(f_ikl[i], c_ic[i], axis=(0, 1))
```

Code Listing 1: Implementation of the streaming step

¹<http://www.numpy.org/>, Version 1.14.3

The collision part of Eq. (9) is also straightforward to implement. To translate an expression like Eq. (9) into vectorized `numpy` code, it can be useful to write it down in Einstein notation. Einstein notation can be cast directly in Python code using the `numpy.einsum` function. Code Listing 2 shows a naive but straightforward implementation of the collision step, split into the computation of the equilibrium distribution function in `equilibrium` and the final collision step in `collide`, where `omega` is the relaxation parameter ω of Eq. (5).

```
import numpy as np
w_i = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])

def equilibrium(rho_kl, u_ckl):
    cu_ikl = np.dot(u_ckl.T, c_ic.T).T
    uu_kl = np.sum(u_ckl**2, axis=0)
    return (w_i*(rho_kl*(1 + 3*cu_ikl + 9/2*cu_ikl**2 - 3/2*uu_kl)).T).T

def collide(f_ikl, omega):
    rho_kl = np.sum(f_ikl, axis=0)
    u_ckl = np.dot(f_ikl.T, c_ic).T/rho_kl
    f_ikl += omega*(equilibrium(rho_kl, u_ckl) - f_ikl)
    return rho_kl, u_ckl
```

Code Listing 2: Naive implementation of the collision step

It is worth pointing out that the collision implementation of Code Listing 2 does not lead to optimal performance. The Python code can be further optimized by unrolling all multiplications with `c_ic` and eliminating the terms that vanish due to zeros in `c_ic`. This implementation is our optimized reference Python implementation. We additionally benchmark this Python implementation against a C++ collision kernel integrated into our Python code with `pybind11`² and `eigen`³.

2.3 Parallelization strategy

We parallelize the LBM using spatial domain decomposition and the message passing interface (MPI)⁴. The collision part of the LBM is then embarrassingly parallel. It is a spatially local operation and no communication is required. Note that we

²<https://github.com/pybind/pybind11>, Version 2.2.2

³<https://eigen.tuxfamily.org/>, Version 3.3.4

⁴<https://www.mpi-forum.org/>

could also have decided to parallelize in the direction space by distributing the occupation numbers for the individual directions onto separate MPI processes. This would then require communication during the collision step. This is, however, disadvantageous because there are just nine directions and parallelism would be limited to just nine parallel processes.

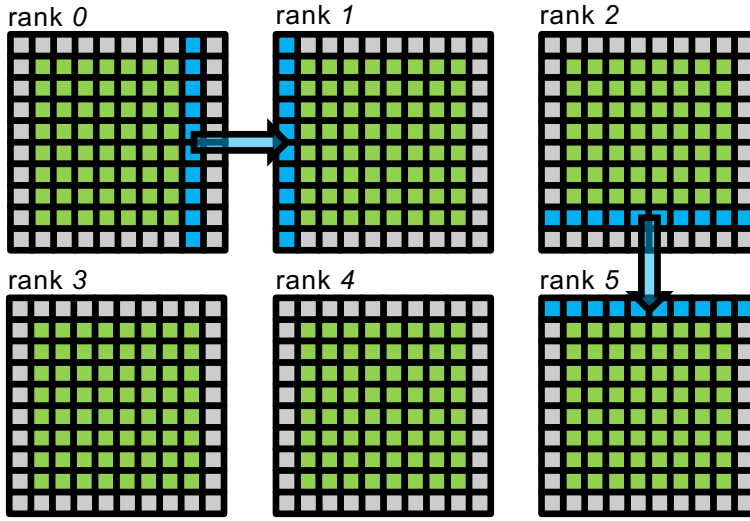


Figure 2: Domain decomposition and communication strategy. We decompose the full two-dimensional lattice into spatial domains of roughly equal size (green lattice points). We then add an additional ghost region of unit thickness surrounding these domains (gray lattice points). During each communication step, we communicate the outermost green active lattice into the adjacent outermost ghost lattice (as exemplified by the blue arrows). This requires four communication steps, two of which are indicated by the arrows.

Parallelization in the spatial domain requires communication during the streaming step. All occupation numbers that are moved past a domain boundary need to be communicated to the neighboring domain. We implement this by adding an additional ghost region to the simulation domain. Figure 2 illustrates the concept for a decomposition in 3×2 domains. The green lattice points are our active computational domain and the grey points are the ghost lattice points. We now communicate the region adjacent to the ghost points into the respective ghost points of the neighboring domain as shown in Fig. 2 *before* the streaming step. Within the streaming step, we then stream the relevant occupation numbers f_i from the ghost points into the active lattice points. This requires a total of four communication steps: Com-

munication to the right and bottom as shown in Fig. 2 (for each domain) plus their reverse, communication to left and top.

We implement communication in Python using the `mpi4py`⁵ bindings to the MPI library. The Python code responsible for communication is shown in Code Listing 3. We need onpe `Sendrecv` call to communicate in each of the four directions. `Sendrecv` takes care of sending data in one direction and simultaneously receiving it from the opposite direction. `ndx` and `ndy` in the code are the number of domains in x and y -direction.

```

from mpi4py import MPI
comm = MPI.COMM_WORLD.Create_cart((ndx, ndy), periods=(False, False))
left_src, left_dst = comm.Shift(0, -1)
right_src, right_dst = comm.Shift(0, 1)
bottom_src, bottom_dst = comm.Shift(1, -1)
top_src, top_dst = comm.Shift(1, 1)

def communicate(f_ikl):
    comm.Sendrecv(f_ikl[:, 1, :], left_dst,
                  recvbuf=f_ikl[:, -1, :], source=left_src)
    comm.Sendrecv(f_ikl[:, -2, :], right_dst,
                  recvbuf=f_ikl[:, 0, :], source=right_src)
    comm.Sendrecv(f_ikl[:, :, 1], bottom_dst,
                  recvbuf=f_ikl[:, :, -1], source=bottom_src)
    comm.Sendrecv(f_ikl[:, :, -2], top_dst,
                  recvbuf=f_ikl[:, :, 0], source=top_src)

```

Code Listing 3: Implementation of the communication step

At the time of this writing, all `mpi4py` communication methods require contiguous `numpy` arrays. The input arrays in Code Listing 3 need therefore be cast into contiguous arrays using `numpy.ascontiguousarray` and a temporary contiguous buffer is required for receiving data. These intermediate steps have been omitted from the code excerpt for brevity.

3 Results for the lid-driven cavity

The lid-driven cavity is frequently used to test fluid dynamics simulation programs. Given a quadratic box with a sliding lid, as shown in Fig. 3a, we use equilibrium

⁵<https://mpi4py.scipy.org/>, Version 3.0.0

initial conditions, Eq. (10), for the discrete BGK-Boltzmann transport equations. The initial values were chosen as $\rho = 1.0$ and $\mathbf{u} = 0$ at time $t = 0$.

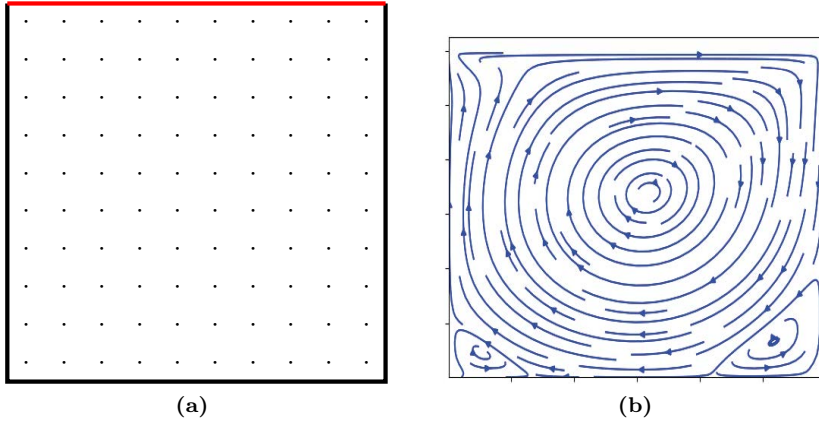


Figure 3: Lid-driven cavity. (a) The system consists of a quadratic box with hard walls drawn in black and a lid that slides to the right with a prescribed velocity, drawn in red. The lattice points all lie inside the box and the walls lie half way between boundary lattice points and (virtual) solid lattice points outside the box. (b) Streamlines in the steady state for a domain of 300×300 lattice points and $\omega = 1.7$, corresponding to Reynolds number 1000, given the typical velocity of the sliding lid as $u_{\text{lid}} = 0.1$. The figures is visualized using the `streamplot` function of the `matplotlib`⁶ library. The lattice is resampled into a 30×30 lattice before determining the streamlines.

The lid moves with a given velocity u_{lid} in direction 1, i. e. to the east. We apply bounce-back boundary conditions on the black boundaries and the prescribed wall velocity boundary conditions on the red wall. The full boundary conditions can be written as

$$f_i = f_{i^*} - 6w_i\rho_{\text{wall}}\mathbf{c}_i \cdot \mathbf{u}_{\text{lid}} \quad (11)$$

where i^* indicates the direction before bounce-back and ρ_{wall} is the fluid density at the wall. More information can be found in Ref. (Mohamad, 2011; Wolf-Gladrow, 2000). Note that we apply bounce back boundary conditions to all parallel domains, but we introduce ghost buffers only for domain boundaries in interior domains, not for the outer boundaries of edge domains. Using this approach, we do not need special treatment of boundary or interior domains. Bounce back of particles in an interior is simply overridden in the next communication step. Outer boundaries of edge domains are not communicated when specifying `periods=(False, False)` in

⁶<https://matplotlib.org/>

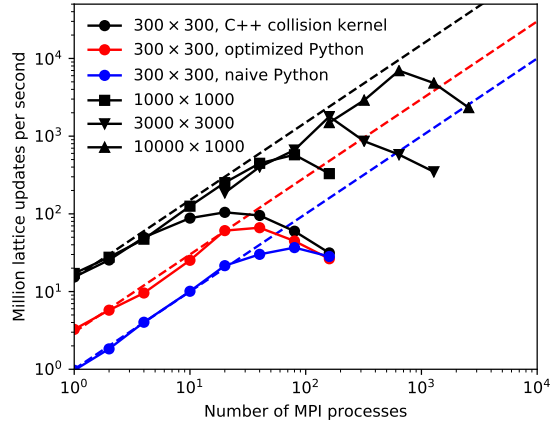
the call to `Create_cart` in Code Listing 3. This keeps our Python implementation as simple as possible and free of specific treatment of corner cases.

We calculated the velocity field in the steady state. Figure 3b shows the streamlines of the flow field after 1 million time steps for a lattice of 300×300 lattice points with $\omega = 1.7$ and the velocity of the lid chosen as $u_{\text{lid}} = 0.1$. The sliding lid induces a large vortex rotating clock wise. The two lower corners show small vortices rotating in the opposite direction.

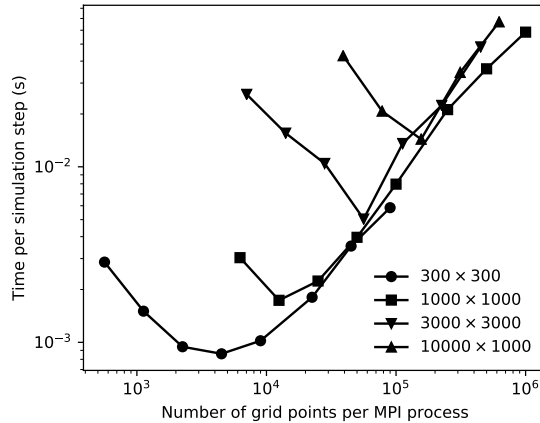
In order to demonstrate the scaling behavior for the parallel version of our code, we calculated the steady state solution of the problem for different sizes of the simulation domain 300×300 , $1,000 \times 1,000$, $3,000 \times 3,000$ and $10,000 \times 10,000$ lattice points on bwForCluster NEMO at the University of Freiburg (2x Broadwell E5-2630v4 at 2.2 GHz per compute node with a 100 Gbit/s Omni-Path interconnect). The square computational lattice was divided in rectangular subregions, with each subregion assigned to one process.

A common measure for the speed of an implementation of the Lattice Boltzmann Method is the number of million lattice updates per second (MLUPS), which we compute for the lid-driven cavity. Fig. 4a reports this measure for the naive and optimized Python implementation as well as for the Python implementation using a C++ implementation of the collision kernel, all using double precision floating-point arithmetics. It is immediately clear that the Python&C++ implementation is fastest, reaching about 15 MLUPS on a single core, while the naive Python implementation reaches only 1 MLUPS. Scaling for all implementations is excellent and the dashed lines in Fig. 4a shows ideal scaling for the three implementations. For a $10,000 \times 10,000$ lattice, the Python&C++ implementation peaks at 7 billion lattice updates per second (BLUPS = 1000 MLUPS) when using 640 MPI process (32 NEMO compute nodes).

Figure 4b shows the execution time (per simulation step) as a function of number of lattice points per MPI process. For each lattice there is a minimum in execution time that moves to a smaller number of lattice points per MPI process with decreasing lattice size. This means the larger the overall lattice, the larger the portions that reside on each MPI process must be for scaling to be optimal.



(a)



(b)

Figure 4: Parallel scaling of the lid-driven cavity on bwForCluster NEMO. (a) Strong scaling test for different system sizes. The dashed lines show ideal scaling with 1 MLUPS, 3 MLUPS and 15 MLUPS per MPI process for the naive, optimized and C++ implementations. (b) Weak scaling test showing the execution time required for a single simulation step for the implementation using C++ collision kernels.

4 Discussion

Figure 4a shows that the three implementations achieve different execution speeds. The optimized Python implementation is already 3× faster than the naive imple-

mentation, while the C++ collision kernels gain another factor of $5\times$ in speed. Scaling is good for all implementations and breaks down at a higher number of MPI processes for the slower implementations. For the 300×300 lattice, all three implementations achieve almost identical execution speed at 160 MPI processes. At this point, the execution time is entirely dominated by the cost of communication between the processes.

Parallel scaling breaks down at different points for different lattice sizes. More insights are obtained from the weak scaling test of Fig. 4b, which shows that there is a minimal number of lattice points per process, such that computation and not communication or idling dominates the aggregate cost. This number depends on the overall size of the lattice and is around 5,000 for the small 300×300 lattice and 100,000 for the largest $10,000 \times 10,000$ lattice. Assuming a square region per process, we end up with a surface to volume relation of about 0.06 to 0.01. Because the point of breakdown occurs at different surface to volume ratios, we believe that cost of communication is not its source. There are slight differences in run-time between the individual processes because the global lattice is not evenly divisible by the lattice used for domain decomposition, leading to slight variations in the size of the local domains. These differences in run-time are likely exacerbated at a larger number of total MPI processes, leading to some idling processes. More investigation is necessary to clarify this point. We also note that this result is specific for the D2Q9 model and is likely different for other two-dimensional and three-dimensional models.

The maximum performance of 7 BLUPS reached by our implementation is comparable to recent reports of 5 BLUPS reached on an implementation on multiple graphics processing units (GPUs) (Xu et al., 2018), albeit for a D3Q19 lattice and a multi-relaxation time collision operation, but (presumably) single precision floating-point arithmetics. Xu et al. (Xu et al., 2018) report 5 BLUPS on 12 NVIDIA K20M GPUs. Assuming linear scaling, we reach 5 BLUPS at 240 MPI processes or 24 NEMO compute nodes. An extensive test of the CPU-based LBM implementation *Musubi* (Hasert et al., 2014) using double precision floating-point arithmetics was recently presented by Qi (Qi, 2017). On Hazel Hen (Haswell E5-2680v3 at 2.50 GHz with a Cray Aries interconnect), *Musubi* achieves around 10 MLUPS per core for a D3Q19 lattice as compared to our 15 MLUPS per core for a D2Q9 lattice.

5 Conclusions

We have demonstrated a parallel implementation of the Lattice Boltzmann Method (LBM) in Python using the `numpy` library and the `mpi4py` bindings to the Message Passing Interface (MPI) that yields excellent scaling on bwForCluster NEMO. We have further shown that implementing the collision operation in the lower level programming language C++ yields a significant performance gain. It eliminates the creation of temporary buffers and avoids multiple loops over data structures. Our C++ optimization appears competitive with recently reported performances of implementations of the Lattice Boltzmann Method (Xu et al., 2018; Hasert et al., 2014), but one has to keep in mind that most published benchmarks are obtained for three-dimensional lattices while we here discuss only the two-dimensional case. Yet, our implementation can still be optimized further, e.g. by fusing streaming and collision steps, looping over data structures in a manner that maintains cache-coherency (Pohl et al., 2003) or by hiding communication behind computation. Future work will focus on extending the present code to three-dimensions and implementing further optimizations.

Implementing high-performance codes in Python has the advantage of fast development times and compact codes that can be used to test implementation and parallelization strategies before optimizing portions of the code in a lower-level language. Our naive parallel LBM implementation has 160 lines of Python code, 34 of which are a parallel implementation of `numpy.save` using MPI I/O. The simplicity of this code makes Python and MPI also suitable for teaching parallel computing. We note that more complex parallel simulation codes implemented largely in Python, for example the GPAW code (Mortensen et al., 2005; Enkovaara et al., 2010) and associated libraries (Larsen et al., 2017) for electronic structure calculations, have emerged over the past years. Python is maturing towards a language that allows rapid development of parallel simulation codes for high-performance computing systems. Libraries such as `pybind11` and `eigen` make extending Python with native numerical code straightforward.

Our full parallel implementation of the Lattice Boltzmann Method can be found online⁷.

⁷<https://github.com/IMTEK-Simulation/LBWithPython>


Acknowledgements


The authors acknowledge the support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG as well as through the Research Unit FOR 2383 ProMiSe under Grant No. GR 2622/6-1.

Corresponding Author

Lars Pastewka: lars.pastewka@intek.uni-freiburg.de
Department of Microsystems Engineering, University of Freiburg,
Georges-Köhler-Allee 103, 79110 Freiburg, Germany

ORCID

Lars Pastewka  <https://orcid.org/0000-0001-8351-7336>

License  4.0 <https://creativecommons.org/licenses/by-sa/4.0>

References

- Aharonov, E. and D. H. Rothman (1993). »Non-Newtonian flow (through porous media): A lattice-Boltzmann method«. In: *Geophys. Res. Lett.* 20.8, pp. 679–682.
- Bhatnagar, P. L., E. P. Gross and M. Krook (1954). »A model for collisionless processes in gases I: small amplitude processes in charged and neutral one-component systems«. In: *Phys. Rev.* 94, pp. 511–525.
- Boltzmann, L. (1896). *Vorlesungen über Gastheorie*. Barth.
- Cercignani, C. (1988). *The Boltzmann Equation and its Applications*. Springer.
- Enkovaara, J. et al. (2010). »Electronic structure calculations with GPAW: A real-space implementation of the projector augmented-wave method«. In: *J. Phys.: Condens. Matter* 22.25, p. 253202.
- Geier, M., A. Greiner and J. G. Korvink (2006). »Cascaded digital Lattice Boltzmann automata for high Reynolds number flow«. In: *Phys. Rev. E* 73, p. 066705.
- Geier, M., M. Schönherr, A. Pasquali and M. Krafczyk (2015). »The cumulant lattice Boltzmann equation in three dimensions: Theory and validation«. In: *Comput. Math. Appl.* 70.4, pp. 507–547.
- Grunau, D., S. Chen and K. Eggert (1993). »A lattice Boltzmann model for multiphase fluid flows«. In: *Physics of Fluids A: Fluid Dynamics* 5.10, pp. 2557–2562.

- Gunstensen, A. K. and D. H. Rothman (1993). »Lattice-Boltzmann studies of immiscible two-phase flow through porous media«. In: *J. Geophys. Res.* 98.B4, pp. 6431–6441.
- Gunstensen, A. K., D. H. Rothman, S. Zaleski and G. Zanetti (1991). »Lattice Boltzmann model of immiscible fluids«. In: *Phys. Rev. A* 43.8, pp. 4320–4327.
- Hasert, M. et al. (2014). »Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi«. In: *J. Comput. Sci.* 5, pp. 784–794.
- Huang, K. (1987). *Statistical Mechanics*. Wiley, New York.
- Larsen, A. H. et al. (2017). »The Atomic Simulation Environment-A Python library for working with atoms«. In: *J. Phys.: Condens. Matter* 29, p. 273002.
- McNamara, G. R. and G. Zanetti (1988). »Use of the Boltzmann equation to simulate lattice gas automata«. In: *Phys. Rev. Lett.* 56, pp. 2332–2335.
- Mohamad, A. A. (2011). *Lattice Boltzmann Method*. Springer.
- Mortensen, J. J., L. B. Hansen and K. W. Jacobsen (2005). »Real-space grid implementation of the projector augmented wave method«. In: *Phys. Rev. B* 71.3, p. 035109.
- Obrecht, C., F. Kuznik, B. Tourancheau and J.-J. Roux (2011). »A new approach to the lattice Boltzmann method for graphics processing units«. In: *Comput. Math. Appl.* 61.12, pp. 3628–3638.
- Pohl, T., M. Kowarschik, J. Wilke, K. Iglberger and U. Rde (2003). »Optimization and profiling of the cache performance of parallel Lattice Boltzmann codes«. In: *Parallel Process. Lett.* 13.04, pp. 549–560.
- Qi, J. (2017). »Efficient Lattice Boltzmann simulations on large scale high performance computing systems«. PhD thesis. Rheinisch-Westflische Technische Hochschule Aachen.
- Succi, S. (2001). *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Clarendon Press, Oxford.
- Wolf-Gladrow, D. A. (2000). *Lattice Gas Cellular Automata and Lattice Boltzmann Models Mechanics*. Springer, Berlin.
- Xu, L., A. Song and W. Zhang (2018). »Scalable parallel algorithm of multiple-relaxation-time Lattice Boltzmann Method with large eddy simulation on multi-GPUs«. In: *Sci. Program.* 2018, p. 1298313.
- Zhang, J. (2011). »Lattice Boltzmann method for microfluidics: models and applications«. In: *Microfluid. Nanofluidics* 10.1, pp. 1–28.