A concise Python implementation of the Lattice Boltzmann Method on HPC for geo-fluid flow

Article *in* Geophysical Journal International · September 2019

DOI: 10.1093/gji/ggz423

CITATIONS
2

READS
1,355

3 authors, including:

Gabriele Morra
University of Louisiana at Lafayette
116 PUBLICATIONS **1,621** CITATIONS

SEE PROFILE

Geophysical Journal International

Geophys. J. Int. (2019) **00,** 1
Advance Access publication 2019 September 26

doi: 10.1093/gji/ggz423

A concise python implementation of the lattice Boltzmann method on HPC for geo-fluid flow

Peter Mora [1], Gabriele Morra [2] and David A. Yuen [3,4]

[1] *College of Petroleum Engineering and Geosciences, King Fahd University of Petroleum and Minerals, Dhahran* 31261, *Saudi Arabia*
[2] *Department of Physics, School of Geosciences, University of Louisiana at Lafayette, LA* 70504, *USA. E-mail: gabrielemorra@gmail.com*
[3] *Department of Applied Physics and Applied Mathematics, Columbia University, New York, NY* 10027, USA
[4] *Department of Big Data, School of Computer Science, China University of Geosciences, Wuhan* 430074, *China*

Accepted 2019 September 24. Received 2019 August 31; in original form 2019 April 4

SUMMARY

The lattice Boltzmann method (LBM) is a method to simulate fluid dynamics based on modelling distributions of particles moving and colliding on a lattice. The *Python* scripting language provides a clean programming paradigm to develop codes based on the LBM, however in order to reach performance comparable to compiled languages, it needs to be carefully implemented, maximizing its vectorized tools, mostly integrated in the NumPy module. We present here the details of a Python implementation of a concise LBM code, with the purpose of offering a pedagogical tool for students and professionals in the geosciences who are approaching this technique for the first time. The first half of the paper focuses on how to vectorize a 2-D LBM code and show how if carefully done, this allows performance close to a compiled code. In the second part of the paper, we use the vectorization described earlier to naturally write a parallel implementation using MPI and test both weak and hard scaling up to 1280 cores. One benchmark, Poiseuille flow and two applications, one on sound wave propagation and another on fluid-flow through a simplified model of a rock matrix are finally shown.

Key words: Permeability and porosity; Geomechanics; Non-linear differential equations; Numerical approximations and analysis; Numerical modelling; Wave propagation.

1 INTRODUCTION

This paper aims to provide a clean Python high performance implementation for the lattice Boltzmann method (LBM) to model fluid flow in geosciences. This method involves simulating the Boltzmann Equations on a discrete lattice—an approach that solves the Navier–Stokes Equations in the macroscopic limit (Frisch *et al.* 1986; Chen & Doolen 1998)—rather than modelling the Navier–Stokes equations themselves. A complete treatise of LBM covering all facets can be found in (Succi 2001). In recent years, the LBM has been applied to various geophysical problems. This includes the study of viscoelastic waves (Xia *et al.* 2017), the study of flow in porous media (Keehm *et al.* 2004; Guo *et al.* 2014), the study of imbibition in porous structures (Zheng & Wang 2018), the study of dissolution and precipitation in porous media (Kang *et al.* 2003; Huber *et al.* 2014), the study of reactive flow in porous media (Kang *et al.* 2010), the study of plumes and convection in the mantle (Mora & Yuen 2017, 2018a,b), the study of melting with convection (Huber *et al.* 2008) and the study of reactive transport (Huber *et al.* 2008; Parmigiani *et al.* 2011). In the following, we will review briefly the LBM prior to presenting the implementation.

The LBM allows fluid dynamics to be modelled by simulating the movement and collision of particle distributions on a discrete lattice in 2-D or 3-D. LBMs have their origins in lattice gas automata (LGA) in which particles move and collide on a discrete lattice representing a simplified discrete version of molecules moving and colliding in a gas. LGA were first proven by Frisch *et al.* (1986) to yield the Navier–Stokes equations in the macroscopic limit. These initial LGA models were unconditionally stable and conserved mass and momentum perfectly. However, they were computationally expensive with averaging needed over space to obtain the macroscopic equations and furthermore, costly calculations were required to evaluate the collision term. Since the initial LGA models, the method has been extended to model distributions

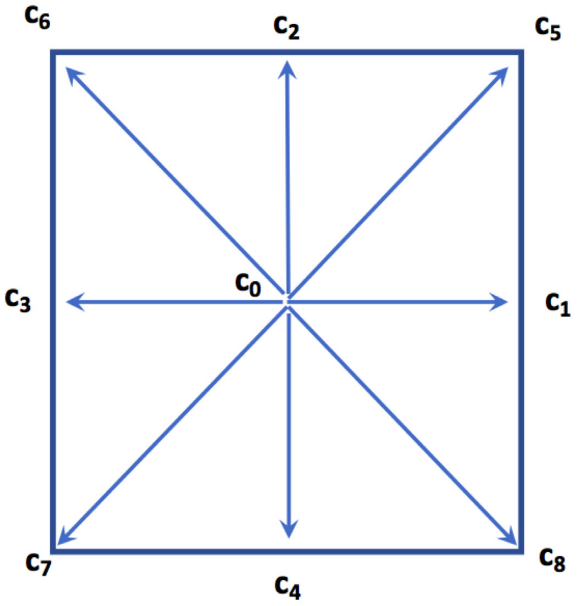

Figure 1. The *D2Q9* lattice.

of particles moving and colliding on a lattice. In these LBMs, one is solving the classical Boltzmann equation on a discrete lattice. An efficient method to calculate the collision term via relaxation was proposed by Bhatnagar, Gross and Krook (BGK, Bhatnagar *et al.* 1954), but only more recently has this been applied to the LBM, enabling efficient algorithms to be developed (e.g. Higuera & Jimenez 1989; Qian *et al.* 1992; Chen & Doolen 1998), and accurate pressure and velocity boundary conditions to be derived (Zhuo & He 1997). Because of the advancement of parallel computing, since the late 1990's, research and applications of the LBM have exploded (see Huang *et al.* 2015; Krüger *et al.* 2017, for a review).

In particular, numerous studies have been conducted of thermal convection (e.g. Shan 1997; He *et al.* 1998; Guo *et al.* 2002; Wang *et al.* 2013; Arun *et al.* 2017). Multiphase methods have been developed as well, such as Shan & Chen (1993), and this remains a highly active research field (Huang *et al.* 2015; Xie *et al.* 2017; Di Ilio *et al.* 2017).

The LBM has been applied to many disciplines of computational science and engineering, due to how well it can be scaled up on high performance computing (HPC) clusters. Recently developed massively parallel software are the Multiphysics WaLBerla (Feichtinger at al. 2011) able to model 10^{12} nodes on Petascale computers, the non-uniform grid implementation reaching up to a trillion grid nodes illustrated in (Schornbaum & Rüde 2016), the widely used Palabos (Lagrava *et al.* 2012), OpenLB (Heuveline & Latt 2007), LB3D (Groen *et al.* 2011), all of them showing excellent performance on HPC. In this work, we illustrate an example of parallel implementation in 2-D, tested up to 1280 cores, only in Python, for pedagogical purposes and for specific applications in geosciences.

This paper provides the details of a technical implementation of the LBM using the *Python* language(Van Rossum *et al.* 2007; Lutz 2013). The goal of this manuscript is to provide geo-scientists and non-experts in parallel programming, with all the details necessary to write their own fast and scalable implementation of the LBM, using only NumPy, the fundamental package with built-in vectorized operations on *N*-dimensional array objects (Morra 2018) and mpi4py, the most commonly used package with the bindings for the Message Passing Interface for Python, (Dalcin *et al.* 2011). Details on how to optimize every segment of the code are described throughout the paper in order to deliver scientists with simple and clear instructions on how to write their own vectorized scalable parallel LBM code using only Python.

2 NUMERICAL SIMULATION METHODOLOGY

The LBM involves simulating particle number densities moving and colliding on a discrete lattice. In one time-step, the particle number densities can move by one lattice spacing along the orthogonal axes, or along diagonals, followed by modification of the number densities at the lattice nodes due to collision. We use the standard notation in LBM denoted *DnQm* for a simulation in $D = n$ dimensions, and with $Q = m$ velocities on the discrete lattice. In the following, we restrict ourselves to 2-D and use the *D2Q9* lattice Boltzmann lattice arrangement shown in Fig. 1. In this lattice, we define $f_\alpha(\mathbf{x}, t)$ as the number density of particles moving in the α-direction where the $Q = 9$ velocities are given by

$\mathbf{c}_\alpha = [(0,0),(1,0),(0,1),(-1,0),(0,-1),(1,1),(-1,1),(-1,-1),(-1,1)]^T$. This choice means that $\mathbf{c}_0$ is the zero velocity vector and so represents stationary particles, and $\mathbf{c}_\alpha = -\mathbf{c}_{\alpha+2}$ for $\alpha = (1, 2, 5, 6)$ are the velocities in the eight directions show in Fig. 1. The lattice is unitary so the lattice spacing and time step are $\Delta x = \Delta t = 1$.

The LBM involves two steps: (a) streaming (movement) and (b) collision of a distribution function. If we wish to model thermal convection or thermal–chemical convection, we just model additional distribution functions, representing the energy and the chemical component (e.g. Luo & Girimaji 2002; Arcidiacono *et al.* 2007; Bartlett 2017). In this paper, we focus on the technical aspects of implementing the LBM using *Python*. We therefore model a single distribution function, f_α representing the mass density of particles moving and colliding in the α-direction on the discrete lattice. The evolution equation encompassing the two steps of moving (streaming) and colliding is given by

$$f_\alpha(\mathbf{x} + \mathbf{c}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) + \Delta f_\alpha^c(\mathbf{x}, t), \tag{1}$$

where $\Delta f_\alpha^c(\mathbf{x}, t)$ is the collision term and represents the redistribution of particle number densities at lattice site $(\mathbf{x}, t)$ due to collisions. The collision term can be calculated exactly, but this is computationally expensive and as such, is rarely done. Alternatively, the collision term can be calculated by the *BGK* method (Qian *et al.* 1992; Chen & Doolen 1998), in which case the distributions relax to the equilibrium distribution. The BGK method is computationally efficient and gives satisfactory results provided the distributions are not too far from equilibrium. The BGK collision term is given by

$$\Delta f_\alpha^c(\mathbf{x}, t) = \left(\frac{1}{\tau_f}\right)(f_\alpha^{eq}(\mathbf{x}, t) - f_\alpha(\mathbf{x}, t)), \tag{2}$$

where $f_\alpha^{eq}(\mathbf{x}, t)$ is used to denote the equilibrium distribution of $f_\alpha(\mathbf{x}, t)$. It should be noted that since the simple single relaxation time BGK collision term given by eq. (2) was proposed, more accurate and stable multiple relaxation time methods have been developed (Lallemand & Luo 2000; *d'Humieres et al.* 2002). The equilibrium distribution is obtained by a Taylor expansion about the Boltzmann distribution given by

$$f^{eq}(\mathbf{u}) = \frac{\rho}{(2\pi RT)^{D/2}} \exp\left(-\frac{(\mathbf{c}-\mathbf{u})^2}{2RT}\right), \tag{3}$$

where D is the number of dimensions. Taking the Taylor's expansion of f^{eq}, we obtain

$$f^{eq}(\mathbf{u}) = \frac{\rho}{(2\pi RT)^{D/2}} \exp\left(-\frac{\mathbf{c}^2}{2RT}\right)\left(1 + \frac{\mathbf{c}\cdot\mathbf{u}}{RT} + \frac{1}{2}\frac{(\mathbf{c}\cdot\mathbf{u})^2}{(RT)^2} - \frac{1}{2}\frac{\mathbf{u}^2}{RT}\right), \tag{4}$$

Noting that the speed of sound is given by $c_s = \sqrt{RT}$, and that on the $D2Q9$ lattice we have $c_s = \sqrt{RT} = c/\sqrt{3}$ where $c = \Delta x/\Delta t = 1$ is the lattice speed, we obtain the equilibrium distribution on the lattice of

$$\begin{aligned} f_\alpha^{eq}(\mathbf{x}, t) &= \rho w_\alpha \left[1 + 3(\mathbf{c}_\alpha \cdot \mathbf{u}) + \frac{9}{2}(\mathbf{c}_\alpha \cdot \mathbf{u})^2 - \frac{3}{2}\mathbf{u}^2\right] \\ &= \rho w_\alpha \left[c_1 + c_2(\mathbf{c}_\alpha \cdot \mathbf{u}) + c_3(\mathbf{c}_\alpha \cdot \mathbf{u})^2 + c_4\mathbf{u}^2\right]. \end{aligned} \tag{5}$$

In eq. (2), the value of τ_f is a relaxation time which relates to the kinematic viscosity ν_f through

$$\tau_f = \nu_f/(c_s^2 \Delta t) + 0.5. \tag{6}$$

In eq. (5) for the equilibrium distribution, the weighting scalars w_α are given by $w_0 = 4/9$ for $\alpha = 0$ (stationary particles), $w_\alpha = 1/9$ for $\alpha = (1, 2, 3, 4)$ (particles travelling along the two Cartesian axes), and $w_\alpha = 1/36$ for $\alpha = (5, 6, 7, 8)$ which are the particles travelling diagonally.

The macroscopic properties, density ρ and velocity $\mathbf{u}$ relate to the distribution function f_α through

$$\rho(\mathbf{x}, t) = \sum_\alpha f_\alpha(\mathbf{x}, t), \tag{7}$$

and

$$\mathbf{P}(\mathbf{x}, t) = \rho\mathbf{u}(\mathbf{x}, t) = \sum_\alpha f_\alpha(\mathbf{x}, t)\mathbf{c}_\alpha. \tag{8}$$

where $\rho = \rho(\mathbf{x}, t)$ is the macroscopic density, $\mathbf{P}(\mathbf{x}, t)$ is the momentum density, R is the universal gas constant, and D is the number of dimensions. In the following, we restrict ourselves to two dimensions ($D = 2$) and we use units such that $R = 1$.

3 A VECTORIZED PYTHON IMPLEMENTATION

3.1 Initializing Python and MPI

Every *Python* program begins with the loading of the packages of relevance to the particular code being developed (Langtangen *et al.* 2006). Implementation of the vectorized version of the code will only use the numerical Python library *NumPy* (McKinney 2012) and the graphical libraries of Python to plot the results: *MatPlotLib*(Hunter 2007):

```
import numpy as np
import matplotlib.pyplot as plt
```

Next, we need to set an array, `ai[]`, that contains pointers to opposite directions and an array, `c[]`, that contains the lattice velocities, and ,`w[]`, the weights associated with each of the nine lattice velocities. Refer to Section 2 for a precise formulation. We use the NumPy module to define the arrays explicitly:

```
c  = np.array([[0,0], [1,0], [-1,0], [0,1], [0,-1], [1,1],
   [-1,-1], [1,-1], [-1,1]]) # Right to left
ai = np.array([ 0, 2, 1, 4, 3, 6, 5, 8, 7])

na = 9   # Number of lattice velocities
D  = 2   # Dimension of the simulation

w0 = 4.0/9.0
w1 = 1.0/9.0
w2 = 1.0/36.0
w  =  np.array([w0,w1,w1,w1,w1,w2,w2,w2,w2])
```

Finally, we need to define physical quantities associated with the relaxation time. This can be done based on a non-dimensional time step `dt` and lattice spacing `dx` equal to 1. Using a constant viscosity value in the entire domain and based on eq. (6), the relaxation time and constants c_1, ...c_4 of eq. (5) can also be computationally defined:

```
dt = 1; dx = 1; S  = dx/dt
c1 =  1.0
c2 =  3.0/(S**2)
c3 =  9.0/(2.0*S**4)
c4 = -3.0/(2.0*S**2)

# Initialize the relaxation time
nu_f         = 0.1        # Viscosity
tau_f = nu_f * 3./(S*dt) + 0.5
```

Next, we must initialize the size of the domains, the number of time steps, and the arrays that we will use. When using NumPy, unlike standard Python, arrays need to be allocated, using the np.zeros() or np.ones() commands, defining their type (Int, Float, etc.). This allows the NumPy routines to quickly access the arrays in a vectorized form and greatly speeds the calculations. The type is generically defined as

`float`, and will be set to either 32 bit or 64 bit float, depending on the architecture of the machine. The size of the grid is `nx×nz`. For this implementation, we initialize nine arrays `f`, `f_stream`, `f_eq`, `Delta_f`, `rho`, `u`, `Pi`, `u2`, `cu`:

```
nt    =  100          # Number of time steps
nx    =   101          # X-axis size
nz    =   101          # Z-axis size

# Initialize arrays
f        = np.zeros((na,nz,nx),dtype=float)
f_stream = np.zeros((na,nz,nx),dtype=float)
f_eq     = np.zeros((na,nz,nx),dtype=float)
Delta_f  = np.zeros((na,nz,nx),dtype=float)
rho      = np.ones((nz,nx),dtype=float)
u        = np.zeros((D,nz,nx),dtype=float)
Pi       = np.zeros((D,nz,nx),dtype=float)
u2       = np.zeros((nz,nx),dtype=float)
cu       = np.zeros((nz,nx),dtype=float)
```

To run an example simulation, it is necessary to initialize the density and velocity. We assume for this simple demonstrative example that there are no obstacles and that the media is homogeneous, and we will show how to add internal heterogeneities. Here the density is stored in `rho` and because the initial speed in the medium `u` is assumed to be zero, the function `f` will depend on the density only, with the appropriate weights. We consider here a point source located in the right of the domain, however the initial density can be modified in any way.

```
# Initialize the density
rho_0               = 1.0          # Density
rho                *= rho_0
rho[nz//2,3*nx//4] = 2*rho_0

for a in np.arange(na):
    f[a] = rho * w[a]
```

Before running a simulation for `nt` time steps, a last step is necessary, which is the creation of a vector of indexes. This is a key passage for running a vectorized simulation. This array of indexes will allow us to apply an instruction in the nine directions of the LBM algorithm and in every `nx×nz` point in a vectorized manner, greatly speeding the streaming calculations by about two orders of magnitude:

```
indexes = np.zeros((na,nx*nz),dtype=int)
for a in range(na):
    xArr       = (np.arange(nx) - c[a][0] + nx)%nx
    zArr       = (np.arange(nz) - c[a][1] + nz)%nz
    xInd,zInd  = np.meshgrid(xArr,zArr)
    indTotal   = zInd*nx + xInd
    indexes[a] = indTotal.reshape(nx*nz)
```

where the array `c[]` that contains the lattice velocities is embedded in a general index `indTotal[]` of dimension `na×(nx×nz)`. This architecture of indexes is what allows us to vectorize the Python version of the kernel of the LBM algorithm.

The Python time loop requires four steps:

1. Imposing the Boundary Conditions (periodic in this example)
2. Calculating the streaming term for `f`
3. Calculating the macroscopic velocity term `u` and the new density `rho`
4. Calculating the equilibrium distribution `f_eq`, based on `u` and `rho`
5. Calculating the collision term `Delta_f` and add it to `f`

The entire loop requires only 20 lines of code:

```
for t in np.arange(nt+1):

    # (1) periodic BC
    f[0:na,0:nz, 0] = f[0:na,0:nz,-2]
    f[0:na,0:nz,-1] = f[0:na,0:nz,1]

    # (2) streaming term
    for a in np.arange(na):
        f_new       = f[a].reshape(nx*nz)[indexes[a]]
        f_stream[a] = f_new.reshape(nz,nx)
    f = f_stream.copy()

    # (3) macroscopic properties: rho and u
    rho = np.sum(f,axis=0)
    Pi = np.einsum('azx,ad->dzx',f,c)
    u[0:D]=Pi[0:D]/rho

    # (4) Equilibrium distribution
    u2 = u[0]*u[0]+u[1]*u[1]
    for a in np.arange(na):
        cu      = c[a][0]*u[0] + c[a][1]*u[1]
        f_eq[a] = rho * w[a] * (c1 + c2*cu + c3*cu**2 + c4*u2)

    # (5) Collision term
    Delta_f = (f_eq - f)/tau_f
    f       += Delta_f
```

We note the following important observations:

1. The `indexes` array created above has been used in the definition of the new collision term `f_new = f[a].reshape(nx*nz)[indexes[a]]`. Here two `reshape()` instructions are used to change shape to the `f` array, but these do not consume any computing time, as they only refer to the internal structure of this array. This is explained in more detail in (Morra 2018, chapter 3).
2. If in the instruction `f = f_stream.copy()`, `copy()` would be absent, then `f` would point to the memory allocated for `f_stream`, which then would be modified.
3. In the instruction `Pi = np.einsum('azx,ad->dzx',f,c)` we used the function `np.einsum()`. This powerful tool allows one to perform any tensorial product, with any high-dimensional array, at the speed of the underlying C optimized code.
4. The instruction `u2 = u[0]*u[0]+u[1]*u[1]` could be written using the Einstein summation function `np.einsum('ijk,ijk->jk', u, u)` or by exploiting the linear algebra library of NumPy `np.linalg.norm(u,axis=0)**2`, but that would not accelerate the code. In the same way `cu = c[a][0]*u[0] + c[a][1]*u[1]` could be written as `np.einsum('j,jkl->kl',c[a],u)` but it would not accelerate the code. For both cases the vectorization of NumPy is equally fast.
5. Separating the instructions `Delta_f = (f_eq - f)/tau_f` and `f += Delta_f` helps to avoid issues with the allocations of the NumPy array `f`.

3.2 Key tools for optimizing the kernel

To illustrate where vectorization plays a role, we show how to write the unoptimized version of the streaming step:

```
for a in range (na):
    for x in range (1,nx-1):
        x_xa = (x - c[a][0] + nx)%nx
        for z in range (nz):
            z_za = (z - c[a][1] + nz)%nz
            f_stream[a][z][x] = f[a][z_za][x_xa]          # Streaming
                step
```

A user might also want to add a level of complexity by implementing regions where waves can penetrate, and others that do not vibrate, either with absorbing or non absorbing BC. If the impenetrable regions are called 'solid', and a `solid` array defines where flow is allowed (zero) and where it is not allowed (one), then the streaming step becomes

```
for a in range(na):
  for x in range(1,nx-1):
    x_xa = (x - c[a][0] + nx)%nx
    for z in range(nz):
      z_za = (z - c[a][1] + nz)%nz
      if solid[z_za][x_xa]:
        f_stream[a][z][x] = f[ai[a]][z][x]    # Bounce-back BC
      else:
        f_stream[a][z][x] = f[a][z_za][x_xa] # Streaming step
```

where the 1D array denoted `ai[]` that was defined previously contains pointers to the opposite directions of flow of the number densities. Use of this pointer array allows so called 'bounce-back' boundary conditions to be applied to solid regions, which is equivalent to a zero-slip boundary condition at the edges of solids.

The code above for the streaming step works but it is extremely slow due to its explicit loops and the `if` statement. The reader can test this implementation and will find a decrease in speed of about two orders of magnitude relative to the optimized streaming step specified below. The same experiment can be done with all the other steps, which have been initially written in this unoptimized manner, and then vectorized.

The optimized streaming step with bounce-back boundary conditions is written as:

```
for a in np.arange(na):
    f_new       = f[a].reshape(nx*nz)[indexes[a]]
    f_bounce    = f[ai[a]]  #bounce back
    f_stream[a] = solid[a]*f_bounce + (1-solid[a])*f_new.
       reshape(nz,nx)
```

where the array `solid[a]` is a Boolean array that is set to `True` if the adjacent point in the `a`-direction is a solid region of the model where particles cannot penetrate, and to `False` if the adjacent point is a fluid region of the model. Here we exploited the equivalency in Python between `0` and `1` and `False` and `True`, respectively.

4 PERFORMANCE OF THE SERIAL CODE

Performance in Python depends on the libraries employed, and by how well optimized its most computationally intensive parts are (see Morra 2018). In particular it is possible to use Just in Time (JiT) compilation on certain routines or functions, and accelerate the code to speeds of 10 per cent slower, or even closer, to standard compiled codes (Behnel *et al.* 2011). Standard vectorization, however, as illustrated in the optimizations employed in the implementation shown in this work, already allows one to accelerate the code from non-vectorized Python by one order of magnitude or more.

We have also performed tests on different computing architectures, in particular, on (i) a MacBook Air with two cores in one 1.3 GHz Intel i5 processor, 8 GB of memory and on (ii) a home PC with 24 processor 2.7 GHz Xeon and 63 GB of memory, for the serial code. For the parallel implementation, we describe later tests using a computer cluster.

For a standard matrix product calculation, we found that on the MacBook Air, the fully vectorized Python version described here was 1.8 times slower than the default compiled (-O) C equivalent, and 3.7 times slower than the fully optimized supported optimization (-O2) of C. On the same machine, by using the Just in Time calculation (Smith 2015) we found that the Python and the compiled C (standard, with -O optimization) where closer, with a gap of less than 10 per cent.

On the PC we found that the vectorized Python was only 10 per cent slower than the (-O) standard compiled version in C, while using the fully optimized (-O2) compilation C was 3.2 times faster than vectorized Python. When using Just in Time compilation on Python, this was about 10 per cent slower than regular (-O) C.

Overall we observe that Python represents a reasonable compromise between performance and usability. Although it can never completely match the performance of C, and is systematically less efficient than the best optimized compiled codes, code development in Python is very clear and easy to write, allowing fast development and debugging. We therefore recommend it for developing new HPC algorithms.

5 PARALLEL VERSION

In the following, we present the algorithm and the *Python* parallel implementation of the LBM. In our implementation the calculations are performed on the slave nodes, while the master node sends and receives data, and creates output files.

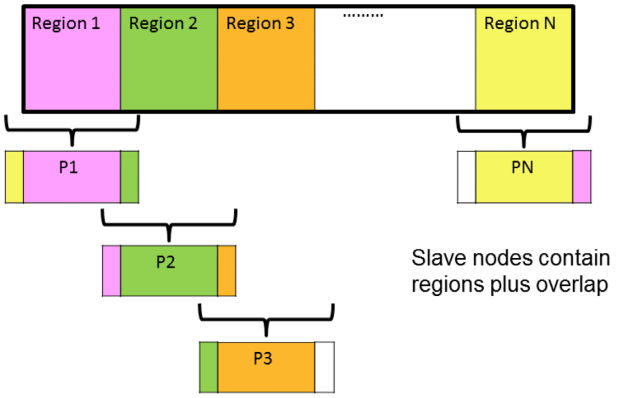

Figure 2. The domain decomposition showing the physical domains modelled in each processor.

5.1 Domain decomposition

The first task of developing an MPI code is to determine how the physical domain maps onto slave nodes. The simplest way is to divide the physical domain into vertical strips. We will assume that the x-axis is longer than the z-axis and choose strips parallel to the z-axis.

Since in the LBM only nearest neighbor communications are needed, in the striped domain decomposition, the portion of data on the slave nodes that has to be communicated covers only a single unit in width. As an example, we show in Fig. 2 the spatial decomposition onto slave nodes and in Fig. 2 the expanded view of a two node problem with $nx = 11$. The first slave node has 6 units of space plus the two overlapping edges, and the second slave node has 5 units of space plus the two overlapping edges. Each domain is a constant size which equalizes work between processors, so the second domain has one empty column. Allowing for the one unit overlap of each domain, each slave node therefore contains seven columns.

5.1.1 Effect of choice of domain decomposition

In order to solve a problem with MPI, one needs to subdivide the problem into domains which are each solved on a processor. The choice of how to subdivide the problem into domains affects the amount of communication required by the algorithm. Here, we derive the formulas for the amount of communication per processor for a domain composition over the x-axis only, and over both the x- and y-axes, and compare these choices. We then generalize to the case of 3-D.

In the following, for simplicity we assume that the calculations are made over a square grid (2-D case) or cubic grid (3-D case) of size n^D. We denote the number of dimensions of the calculations as D and the number of dimensions for the domain decomposition as d.

Consider first, a 1-D domain decomposition ($d = 1$) in a 2-D domain ($D = 2$). In this case, the amount of communication per processor denoted $C(d, D)$ is proportional to twice the size of the grid

$$C(d = 1, D = 2) = 2n, \tag{9}$$

where the factor of 2 is due to having two sides to the domains. Hence, the communication cost per processor is proportional to the length of the y-axis for a problem that divides the x axis onto the n_p processors.

Now, we calculate the communication cost for the 2-D problem ($D = 2$) decomposed into square domains spanning onto the n_p processors ($d = 2$). The area of each domain is given by

$$A = n^2/n_p.$$

Hence, the length of each side of the domain is given by

$$L = \sqrt{n^2/n_p} = n_p^{(-1/2)}n.$$

In this case, the communication cost is proportional to four times the length of the square domains. The factor of 4 is due to there being two sides of each domain along each of the two dimensions. Hence, the communication cost is proportional to

$$C(d = 2, D = 2) = 4L = 4{n_p}^{-1/2}n, \tag{10}$$

One observes that the domain decomposition over $d = 1$ dimensions is less costly by a factor of 2 relative to the $d = 2$ dimensional domain decomposition, but more costly by a factor of $n_p^{1/2}$. Hence, for small problems (eg. $n_p \sim 1$), the $d = 1$ dimensional domain decomposition is less costly by up to a factor of 2, while when $n_p \gg 1$, the $d = 1$ dimensional domain decomposition becomes more costly than the 2-D domain decomposition ($d = 2$) by a factor of $\sqrt{n}_p$. The number of processors when the two domain decompositions are equally efficient can be calculated by solving

$$n = 2n_p^{-1/2}n,$$

which yields a crossover point at

$$C(1,2) = 2n = C(2,2) = 4n_p^{-1/2}n \Rightarrow n_p = 4. \tag{11}$$

In other words, for small cases with $n_p < 4$, the 1-D domain decomposition is most efficient, but the 2-D domain decomposition rapidly becomes much more efficient for cases of $n_p > 4$, and tends towards being more efficient by a factor of $\sqrt{n_p}/2$.

Next, we consider the communication cost for a 3-D problem. The cases for a 1-D and 2-D domain decompositions ($d = 1, 2$) are already covered by the - example above, aside from an additional factor of n required to calculate the area of domains (i.e. the area of the side of domains is given by $L \times n$ where L is the length of the domains on the $x-y$ plane, and the factor n is the length of domains over the z axis). Namely, we have the communication cost for the case of $D = 3$ for 1-D and 2-D domains ($d = 1, 2$) given by

$$C(1,3) = 2n^2, \tag{12}$$

and

$$C(2,3) = 4n^2 n_p^{-1/2}. \tag{13}$$

Next we consider the case of a 3-D problem ($D = 3$) decomposed into 3-D cubes ($d = 3$). The volume per processor is given by

$$V = \frac{n^3}{n_p},$$

so the length of the sides of the domains is given by

$$L = \left(\frac{n^3}{n_p}\right)^{-1/3} = n_p^{-1/3}n.$$

The area of sides of the domain cubes is L^2, and each face of the cubes must be communicated (6 faces of a cube), so the communicational cost is proportional to

$$6L^2 = 6n^2 n_p^{-2/3}. \tag{14}$$

The above formulae can be generalized to work for all of the above cases (i.e. $d = 1, \ldots, 3$, $D = 2, 3$). The generalized equation for communication costs is given by

$$C(d, D) = 2dn^{(D-1)}n_p^{(1-d)/d}. \tag{15}$$

Hence, the communication cost for a 3-D problem ($D = 3$) using a 3-D relative to a 2-D domain decomposition ($d = 3$ relative to $d = 2$) is given by

$$\frac{C(3,3)}{C(3,2)} = \frac{3n_p^{-2/3}}{2n_p^{-1/2}} = \frac{3}{2}n_p^{-1/6}, \tag{16}$$

As for the previous 2-D case ($D = 2$) for 1-D and 2-D domain decompositions ($d = 1$,2), for a small number of processors ($n_p \sim 1$), the relative cost of the $d = 2$ domain decomposition is faster by a small factor (3/2). And as the number of processors increases, the $d = 3$ domain decomposition is faster by a factor of $n_p^{1/6}$ ($n_p \gg 1$). The crossover point where the $d = 3$ domain decomposition becomes more efficient than the $d = 2$ decomposition is given by

$$n_p^{1/6} = (\frac{3}{2}) \Rightarrow n_p = \left(\frac{3}{2}\right)^6 \approx 11. \tag{17}$$

The generalized equation for the crossover when it becomes more efficient to use a d-dimensional domain decomposition compared to a ($d - 1$)-dimensional decomposition can be solved by setting $C(d, D)/C(d - 1, D)$ to unity. Hence, we have

$$n_p^{crossover} = \left(\frac{d}{(d-1)}\right)^{d(d-1)}.$$

In other words, once the number of processors exceeds $n_p = 11$, the 3-D domain decomposition is more efficient than a 2-D domain decomposition, and is much more efficient than a 1-D domain decomposition. We conclude that it is most efficient to use a domain decomposition over all axes (i.e. $d = D$). However, to simplify the example and `Python` coding in this didactic paper, we use a 1-D domain decomposition in the following 2-D examples. Also, we find that communication costs are small relative to other parts of the LBM algorithm. A future paper is planned on the implementation of the parallel 3-D LBM for convection.

5.2 Initializing of MPI

Besides loading the numerical Python library, we have here to load the MPI libraries, and initialize them in order to know the number of nodes (*size*) and on which node (*rank*) the software is presently running:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
```

Next, we must initialize the size of the domains. Let us suppose that the size of the grid is `nx×nz`. We then partition the grid as even widths along the *x*-axis as is possible. Hence, we will have that the number of domains is `nr=nx/nx_i` where `nx` is the total size of the *x*-axis, and `nx_i` is the size of a domain which must include the overlap of the adjacent domains. The size of `nx_i` can be therefore calculated using:

```
nx_i = (nx+(size-1)*2)//(size-1)
while (nx_i-2)*(size-1)<nx: nx_i += 1
nxi = nx_i - 2 ;
```

which first estimates the size of `nx_i` and then updates it to ensure the entire *x*-axis can fit into the `(size-1)` slave nodes. Note that in the above, `nxi` is the size of the domain exclusive of the two overlap columns.

Next, we need to calculate how many slave processors out of the total number of `(size-1)`, can be used to model the entire domain of size `nx`. This is achieved as follows:

```
nr  = nx//nxi
dnx = nx%nxi
if dnx > 0: nr += 1
nr += 1
```

where if `dnx>0`, then `dnx` is the width exclusive of the overlap columns of the final `(nr-1)`th active slave node. In the above, `nr` denotes the rank or total number of processors that are active including the master node. Hence, processor 0 is the master node and processors 1 through `(nr-1)` are slave nodes.

It should be noted that due to the fact that we divide the domain into integer widths that are located on an integer number of processors, sometimes the `(nr-1)`th node may only span `dnx<nxi` columns of physical space. Furthermore, the number of nodes to span the physical space may be less than the number of processors being used, that is `(nr-1)<(size-1)`. This is unavoidable but means that sometimes for a given number of processors, these cannot all be used to model the `(nr-1)` regions of space that span the total physical space. If we denote `mx` as an array specifying the size exclusive of overlap of each of the `(nr-1)` active slave nodes, we have:

```
mx = np.zeros(size,dtype=int)
for r in range(nr): mx[r] = nxi
if dnx > 0: mx[nr-1] = dnx
```

5.3 Initializing the model in the domains

In Python, arrays are only allocated and not deallocated because the *Python Runtime*, through the garbage collector, automatically deallocates the arrays through evaluating when they are not used anymore and therefore deallocates them when required. This is one of the advantages that makes developing Python codes easier.

Let's look now at how to allocate and set the values of arrays on the master node and on the slave nodes. In the following, we utilize `rank=0` as the master node on which we initialize the model in the entire model space, and utilize `rank>0` to specify each domain of the domain decomposition. Thus, the density ρ (specified as `RHO`), the velocity u (specified as `U`), and the definition of the non deformable parts of the model (specified as `SOLID`) in the entire 2-D space of size $nz \times nx$ can be initialized on the master node with the following:

```
nx = 1001
nz = 251
if rank == 0:
        SOLID  = np.full((nz,nx), False)
        RHO    = np.ones((nz,nx))
        U      = np.zeros((D,nz,nx))
```

where in the above, we specified first the size of the rectangular box where we run the simulations as, for example, $nx = 1001$ and $nz = 251$.

Once all of the variables that span the physical space are initialized on the master node (i.e. `rank=0`), namely, the density denoted `RHO` and the velocity denoted `U`, one must send the nth domain of `RHO` and `U` to the nth node (i.e. `rank=n`). The following code segment achieves the process of sending each domain of space to the appropriate node and sets an array `Solid[a]` which defines whether the adjacent point in direction α is solid or fluid. Subsequently, we specify routine `put_rho_u()` which puts the density and velocity onto nodes, and mention

`put_solid()` which similarly puts the solid array `SOLID` onto the nodes as array `solid`, where `SOLID` is `True` in solid regions of the model, and `False` in fluid regions.

```
put_rho_u()
put_solid()
for a in np.arange(na):
    for z in np.arange(nz):
        for x in range(nx_i):
            if solid[_z_za_[a][z]][_x_xa_[a][x]]:
                Solid[a][z][x] = True
            else:
                Solid[a][z][x] = False
```

Note that in the present implementation, the blocks sent using MPI are large (the entire interior of each domain associated with each processor). This delegates the effort of optimizing the MPI communication to the `mpi4py` library. An alternative would be to send the columns of each domain one by one. This choice is machine and problem-dependent.

In the *Python* code shown in this paper, uppercase `RHO` and `U` are respectively used to denote the density and velocity on the master node (`rank=0`) and lowercase `rho` and `u` are, respectively, used to denote the density and velocity on the slave nodes (`rank>0`). Note that the routine `put_rho_u()` also initializes the number densities denoted `f` using the equilibrium distribution shown in eq. (5), exactly as was done in the serial version:

```
def put_rho_u():
    if rank == 0:
        for r in np.arange(1,size): #iterate through the nodes
            x0=(r-1)*nxi #memory chunk

            # send densities to all nodes
            tmpRho = RHO[0:nz,x0:x0+mx[r]].copy()
            comm.Send([tmpRho, MPI.FLOAT],dest=r)

            # send velocities to all nodes
            tmpU = U[0:D,0:nz,x0:x0+mx[r]].copy()
            comm.Send([tmpU.MPI.FLOAT],dest=r)

    if rank > 0 and rank < nr:
        # receive densities at the node "rank"
        tmpRho = comm.Recv(source=0)
        rho[0:nz,1:mx[rank]+1]=tmpRho

        # receive velocities at the node "rank"
        tmpU = comm.Recv(source=0)
        u[0:D,0:nz,1:mx[rank]+1]=tmpU

        # initialize the distribution function f
        u2 = np.einsum('ijk,ijk->jk', u, u)
        for a in np.arange(na):
            f[a] = rho * w[a] * c1
            cu = np.einsum('i,ijk->jk', c[a], u)
            for d in np.arange(D):
                f[a] += w[a]*(c2*c[a][d]*u[d] + c3*cu**2 + c4*u2)
```

The routine `put_solid()` is written similarly, but taking into account that it is a Boolean array.

5.4 Communication of Python objects

MPI for Python supports two types of communication.

1. A `pickle-based` communication of generic Python objects, using standard commands `send()`, `recv()`, `bcast()`. This option is very easy to use, as it does not require any type or object size specification. In order to send buffer objects, the receiving array must be sufficiently large. This option is the slowest, as it is not designed to be optimized for NumPy arrays.

2. A dedicated communication of `buffer-like` objects. This is designed to be based on the same commands as the `pickle-based` commands, but with a capitalized first letter of the commands, for example `Send()`, `Recv()`, `Bcast()`, `Scatter()`, `Gather()`. In this case, the data types must be specified in both the sending and receiving commands, for example `MPI.INT` and `MPI.FLOAT`.

Because of its greatest efficiency and specific design for NumPy arrays, we use `buffer-like` instructions in the following (i.e. commands with capitalized first letters).

When designing the parallel implementation of a parallel code, one has to choose between *Point-to-Point* and *Collective* communication. While for very large problems, it is more efficient to use the *Collective* options (e.g. Broadcasting, Scattering) we prefer to present here an implementation based on *Point-to-Point* instructions. This is based on (i) the observation that the Communication Costs are minimal compared to the Computing Time, at least up to about 1000 processors, for which we made our tests and (ii) the greater flexibility of a *Point-to-Point* implementation, which allows one to send blocks of different sizes for domains that are not a simple multiple of the number of cores.

Another design choice is between *blocking* and *non-blocking* instructions. Blocking commands block the program until the data have been sent to the destination node, and the buffer is available again. This approach is easier to use, but it implies a greater role played by the local MPI implementation. On some Systems, MPI may save the data freeing immediately the buffer, on others instead the data have to first reach the destination. We tested both *blocking* and *non-blocking* communication and found comparable performances on the Beowulf system on which we performed our tests. On some systems *non-blocking* communication might be required. In this case it will be necessary to use `ISend()` and `IRecv()` to send and receive data, and the `Wait()` call to check whether the communication has finished. In Section 6, we show that the time required for communication is several orders of magnitude less than the total computing time, which implies that *blocking* instructions are sufficient. For larger system, however, a more sophisticated *non-blocking* implementation might be required. For pedagogical and simplicity reasons, we also believe that the present implementation is more suitable to help the reader who is new to parallel programming.

5.5 Getting the edges of each domain in MPI

Each slave node contains a segment of physical space plus an edge from adjacent slave nodes to the left and right to allow the nearest neighbor communications needed for the LBM streaming step. The standard *Python* command to send an edge to an adjacent slave node is of form `comm.send (edgeR, dest=dest_node)` for the right edge, and `comm.send (edgeL, dest=dest_node)` for the left edge. For each signal being sent, there must be a command to receive the message from each slave node of form `edge = comm.recv( source = source_node) .`

The above are the standard Python commands, valid for any data type. However, we show here the faster, vectorized commands designed for NumPy arrays. The instruction differs because it starts with a capital letter (`Send` instead of `send`, and `Recv` instead of `recv`), and because it exploits the declaration of the variable type (`MPI.FLOAT` in this case). Here follows the *Python* code needed to send the two edges of each slave node to the right and left, and to receive these two edges from the adjacent nodes.

```
def get_edges():
    if nr == 2:  # if there is only one slave cpu
        if rank == 1:
            f[0:na,0:nz,0      ] = f[0:na,0:nz,mx[1]]
            f[0:na,0:nz,mx[1]+1] = f[0:na,0:nz,   1 ]
    else: # any other number of slave cpus
        if rank > 0 and rank < nr:
            edgeR = np.zeros(nz)
            edgeL = np.zeros(nz)

            rr = rank+1 if rank<nr-1 else 1  #right block
            rl = rank-1 if rank>1 else nr-1  #left block

            for a in range(na):
                comm.Send([ (f[a,0:nz,mx[rank]]).copy(), MPI.FLOAT
                    ] ,dest=rr)
                comm.Send([ (f[a,0:nz,1]).copy(), MPI.FLOAT ],
                   dest=rl)

                comm.Recv(edgeR,source=rr); f[a,0:nz,mx[rank]+1] =
                    edgeR[0:nz]
                comm.Recv(edgeL,source=rl); f[a,0:nz,0] = edgeL[0:
                   nz]
```

Notice the instruction `copy()` after the slice of the array `f()`: this allows reallocating that slice into a contiguous array, which is necessary when using the fast communication features. Note how the right and left edges use two different allocations for the array, as for

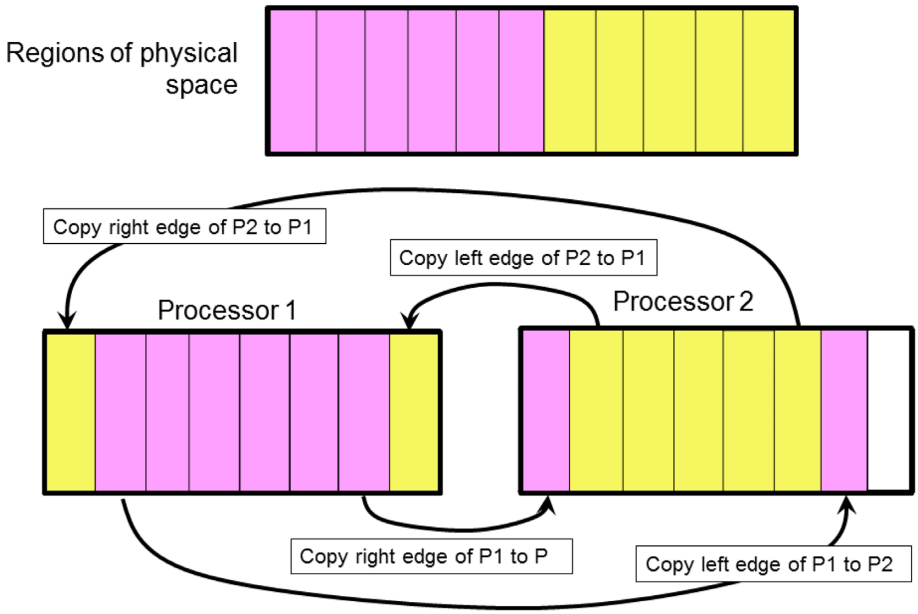

Figure 3. An enlargement of a domain decomposition showing two physical domains modelled in two processors and the circular boundary conditions.

every slave node they have to be stored separately. This also helps accelerate the synchronization (left and right edges are sent first together, and then received together) and therefore improves the code performance (Fig. 3).

5.6 Setting the boundary conditions

In the examples shown in this paper, we specify a simple model of a porous solid rock matrix (Torquato 2013), and simulate (i) the flow from the left to the right of the model due to a density (and hence pressure) difference at the left and right boundaries and (ii) the propagation of a wave front through the matrix, where the grains are assumed rigid, and are therefore reflective for the incoming wave.

The following code shows the *Python* implementation that sets simplified left and right boundary conditions by using the left and right density to calculate the zero velocity equilibrium distributions. It should be noted that setting of accurate pressure or velocity boundary conditions is much more involved (Zhuo & He 1997) and is not shown here where the main purpose is to show the *Python* implementation of the core LBM. As the left boundary is located on node 1 (i.e. `rank=1`) and the right boundary is located on node `(nr-1)` (i.e. `rank=(nr-1)`), we require no MPI communication step to set the densities at these boundaries. For approximate general boundary conditions at any location in space, a code similar to the `put_rho_u()` routine using the equilibrium distribution to set the number densities can be written (see Section 5.3), although again, it should be noted that accurate pressure or velocity boundary conditions require a more detailed treatment (Zhuo & He 1997):

```
def put_rho_boundaries(rho_left,rho_right):
    if rank == 1:
        x = 0
        f[0:na,0:nz,x+1] = np.outer(w[0:na],rho_left*np.ones(nz))
    elif rank == nr-1:
        x = mx[rank]
        f[0:na,0:nz,x+1] = np.outer(w[0:na],rho_right*np.ones(nz))
```

Although the gain is minimal, by calling this routine only once for each loop of the simulation, we vectorize the assignment and compacted it into one line of code for each case. Notice how the float `rho_left` and `rho_right` are mapped to an array through `np.ones(nz)` in order to match the dimensionality of the boundary. The `np.outer` instruction represents an *outer product*, which allows two 1-D arrays to be combined into one 2-D array.

5.7 Getting the density for plotting

In order to plot the results, which is managed from the master processor, one must get the density from the slave processors back to the master processor. The following Python code gets the density from slave processors back to the master processor. This

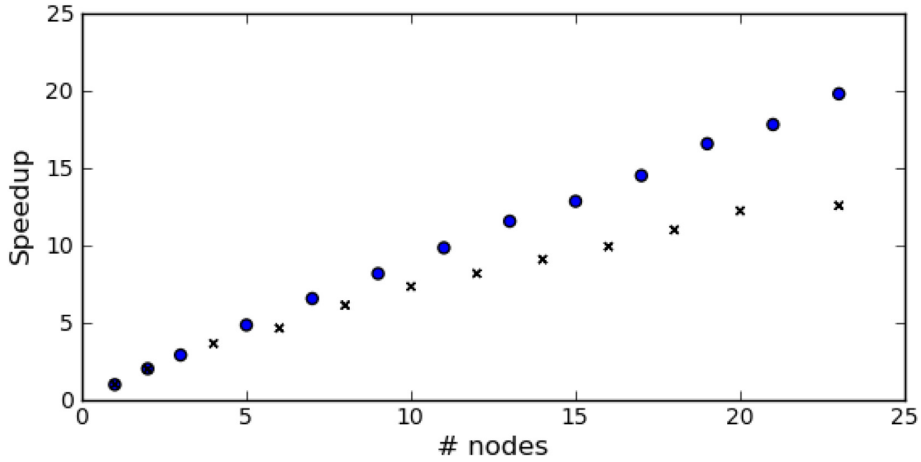

Figure 4. The speedup as a function of processor with up to 24 processors (o = 1001 × 1001 case, × = 101 × 101 case).

operation is vectorized using NumPy arrays (McKinney 2012) and uses the associated vectorized `MPI` instructions from it mpi4py (Dalcin *et al.* 2011).

```
def get_rho():
    if rank > 0 and rank < nr:
        comm.Send([rho,MPI.FLOAT],dest=0)
    if rank == 0:
        for r in range(1,nr):
            tmp = np.zeros((nz,1+mx[rank]+1))
            comm.Recv(tmp, source=r)
            x0=(r-1)*nxi
            RHO[0:nz,x0:x0+mx[r]] = tmp[0:nz,1:1+mx[r]]
```

6 PARALLEL PERFORMANCE

The parallel code was benchmarked on two different machines. One home system running up to 24 processor 2.7 GHz Xeon with 63 GB of memory. On this machine, the *Python* version was V2.6.6, installed on Linux Centos V 6.6. Fig. 6 shows the speedup as a function of processor for a 2-D model of sizes 101 × 101 and 1001 × 1001 (Fig. 4).

A second test was performed on the Queen Bee 2 Beowulf Cluster, a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10 000 Intel Xeon processing cores, 56 Gb/sec (FDR) InfiniBand 2:1 and 1 Gb s^{-1} Ethernet management network. We tested the code on 1, 2, 4, 8, 16, 32, 48 and 64 nodes, corresponding to 20, 40, 80, 160, 320, 640, 960 and 1280 processors, respectively. Given the size of the computing tools, we tested the model with an increasing number of computing points. We ran 5000 timesteps on a grid of size 20 000 points on the z-axis and $(N_{CPUS} - 1)*25$ on the x-axis, where N_{CPUS} is the number of processors. With this choice of parameters, the problem size increases linearly with the number of cpus, ranging from 20 000 × 475 points on 20 processors up to 20 000 × 31 975 (over 600 millions computing points) on 1280 processors. To use $N_{CPUS} - 1$ as a parameter was due to the fact that the domain is divided among all the processors except the first (master core).

Fig. 5 shows the speedup as a function of the number of processors on the Beowulf cluster for both the weak scaling illustrated above and strong scaling of a problem of 20 millions computing points (10 000 × 2000) (Numrich 2018) (Chapter 6). The plots for the speedup are shown in log–log scale (left) and allow the scaling to be estimated over several orders of magnitude in numbers of processors. On the right, we show the normalized performances on linear-linear scale (i.e. the speed divided by the number of processors normalized by the smallest case, 20 cpus). Weak scaling results are shown as dots and are based on the numbers shown in Table 1. The scaling is perfectly linear up to 960 processors, and shows a slight decrease in performance for the case of 1280 cpus. The results for the strong scaling problem are more complex. We observe a threshold number of cpus above which the solver scales linearly with the number of processors, which varies with the problem size. For 20 millions computing points, the threshold is at about 200 cpus. This behavior is likely related to the size of the cache involved into handling the large arrays in the MPI implementation.

To understand the role of each component of the code on the performance, we compiled a detailed table with the broken down computing time for the two communications necessary at every step (*get_edges()* and *put_rho_boundaries()*), and of the four computing steps—(1) streaming, (2) calculation of the macroscopic properties ρ and u from the number densities f_α, (3) calculation of the equilibrium number densities f_α^{eq} and (4) calculation and addition of the collision term Δf_α^c. The four computing steps are respectively denoted *Stream, Macro, Equilibrium and Collision*. The exact timing for all of these steps for all the simulations for all the parallel tests are shown in Table 1. The

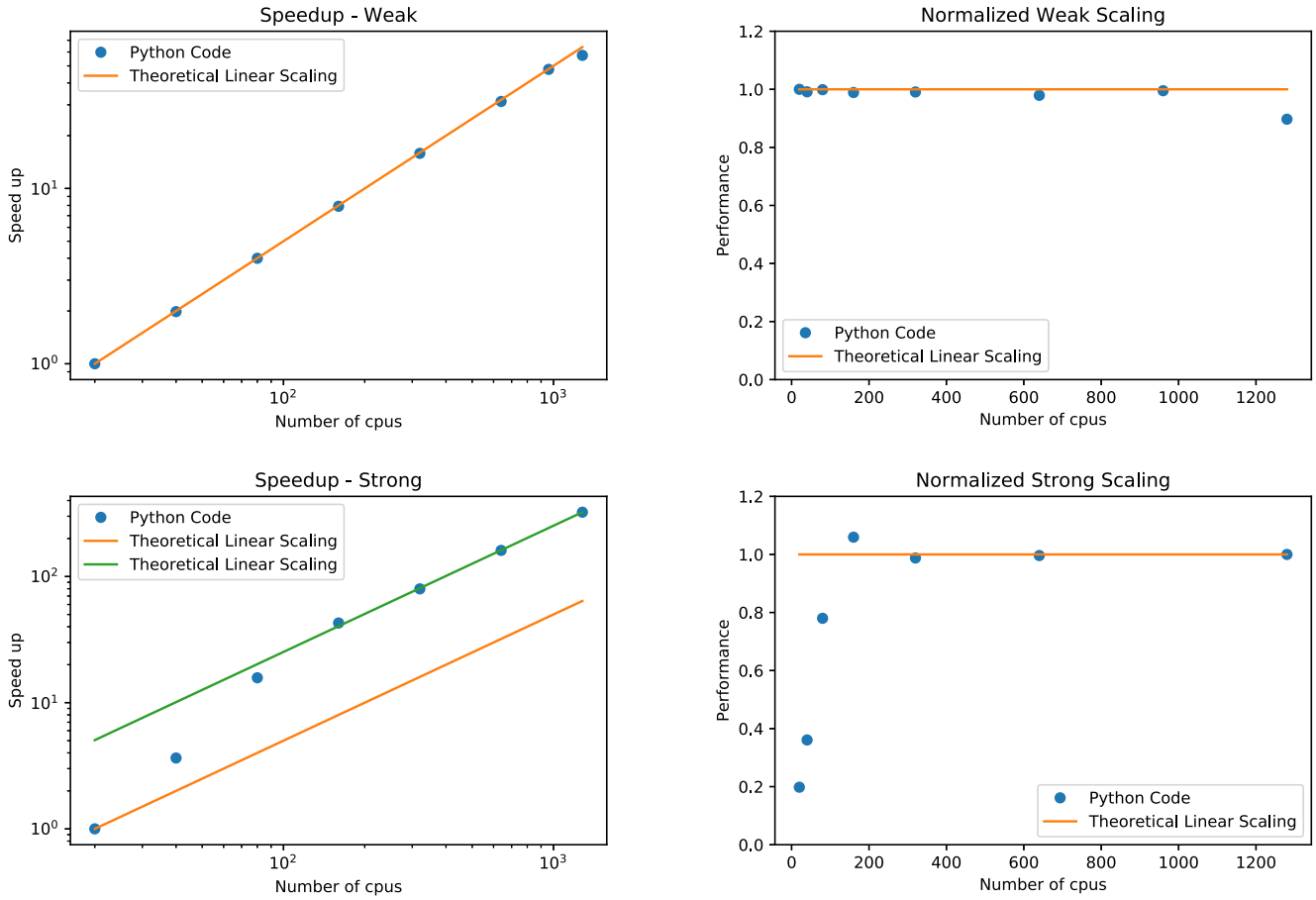

Figure 5. Weak and Strong Speedup and Scalings versus number of processors up to 1280 processors. Top left-hand panel: log–log plot of the speedup for the Weak scaling case. Top right-hand panel: linear–linear plot of the normalized performance, always assuming 1 for one node (20 cpus). One observes a slight decrease only for 1280 processors. On the bottom the plots show the same for Strong scaling, i.e. same problem size regardless to the number of processors. In this case for 20 millions computing points (10 000 × 2000) the code starts scaling linearly from 160 cpus up to 1280. For a smaller number of processors, there is not enough cache to properly handle the large arrays.

Table 1. Timing table for the weak scaling test of the parallel code, broken down to its detailed components. Times are in percentage of the total time, for the weak problem described in the test (size growing proportionally to the number of processors). The performance remains stable from 20 up to 1280 processors, with only a slight decrease in performance at 1280 processors, due to an increase in the time taken by the computational steps of the algorithm, mainly *Collision*.

# Cpus	Get edges (per cent)	Put rho bound (per cent)	Stream (per cent)	Macro (per cent)	Equilibrium (per cent)	Collision (per cent)
20	0.002	0.002	30.936	15.655	34.244	19.162
40	0.001	0.001	30.776	15.625	34.316	19.280
80	0.001	0.001	30.752	15.611	34.202	19.432
160	0.001	0.001	30.890	15.547	34.215	19.346
320	0.001	0.001	30.636	15.562	34.328	19.471
640	0.002	0.002	31.099	15.477	34.528	18.892
960	0.001	0.002	30.894	15.570	34.194	19.339
1280	0.001	0.001	30.785	14.811	32.689	21.713

most important observation is that the time required for all the communications is several orders of magnitude smaller than the computing time, meaning that the algorithm that we present is not prone to deadlocks or slow down for any number of processors. This can be appreciated clearly in the left-hand part of Fig. 6 where the times of the two communication steps are combined into one. This log-linear plot illustrates the orders of magnitude difference between computing and communication times, and demonstrates that communication costs are negligible.

A more detailed look at the computing times, on the right of the Fig. 6, finally shows that the slight decrease in performance at 1280 processors is due mainly to an increase in the computing time of the *Collision* step, and in part to the other three computing steps. Overall, the communication is optimal enough for practical uses and also for the large problems considered here.

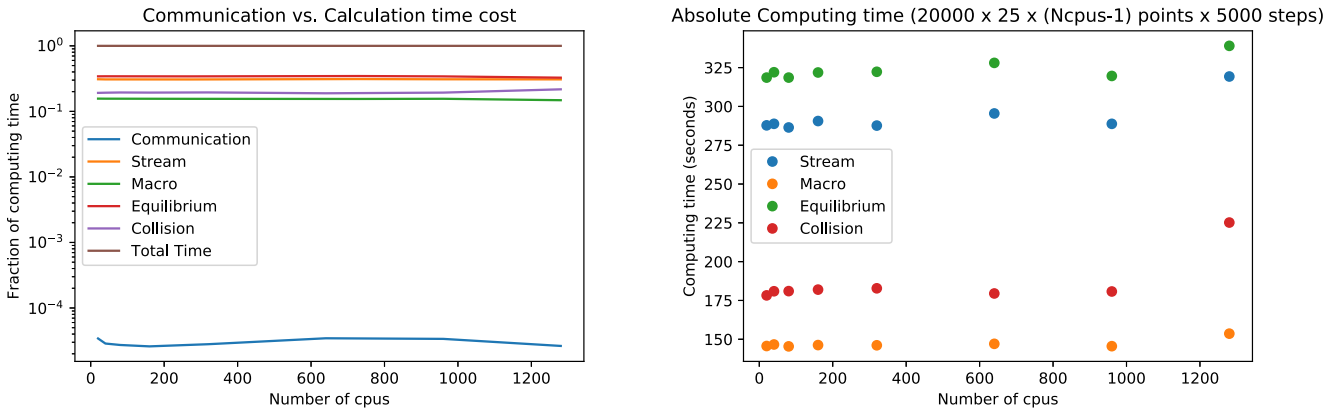

Figure 6. Performance for the parallel code, broken down in its parts, the four computational steps (*Stream, Macro, Equilibrium and Collision*), and the combined cost of the two communications steps (*Communication*).

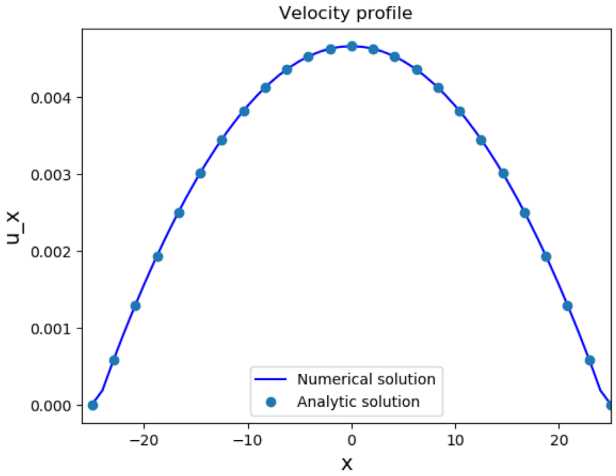

Figure 7. Benchmark results for the Poiseuille flow.

7 APPLICATIONS

The code has been benchmarked with Poiseuille flow, a common test in the geodynamic literature (Gerya 2009). Fig. 7 shows that the numerical solution matches the analytical solution on the entire domain. Towards the edges, the kink is due to the bounce-back Boundary Conditions which are at half way between lattice sites (not exactly at lattice sites).

Simulations with this code have been performed for acoustic wave propagation and for fluid flow, both in a simplified solid rock matrix made up of circular grains and grain clusters. The matrix used for the fluid flow simulation is shown in Fig. 8 (left), and the matrix used for the wave propagation simulation is shown on the same figure, to the right.

For the fluid-flow dynamic simulations, we have set the density of the fluid in the pore space to unity, with the left and right boundary, respectively, having a non-dimensional density of 1.01 and 0.99, respectively. These values have been chosen so that the differential density is a very small compared to the absolute magnitude. The result for speed of the fluid in the simulation after 5000 time steps is shown in Fig. 9.

It is well known that acoustic waves can propagate inside fluids in a porous medium (Lighthill 1978). The second simulation example for acoustic wave propagation is a simplified case of waves inside fluids of a porous medium. The results of the acoustic propagation simulations are shown in Fig. 10, where a wave propagates from an initial point in $nz/2$, $3/4{*}nx$. The wave front requires about 330 steps on a resolution of 401 $\times$ 401 to cross the entire domain and interact with the propagating wave front through periodic boundary conditions. Snapshots of fluid density in the simulation in this figure are shown up to time step 400.

8 CONCLUSIONS

The LBM is a flexible computational tool that allows, among other things, one to calculate wave propagation and fluid flow in complex strongly heterogeneous media. We show how a code developed entirely in Python displays exceptional performance, only inferior to the best optimized compiled C, if carefully written in a vectorized form (Morra 2018), and with the use of Just in Time compilation for selected functions. Compared to C and Fortran, however, Python is easier to write (Guttag 2013), to understand, and to debug.

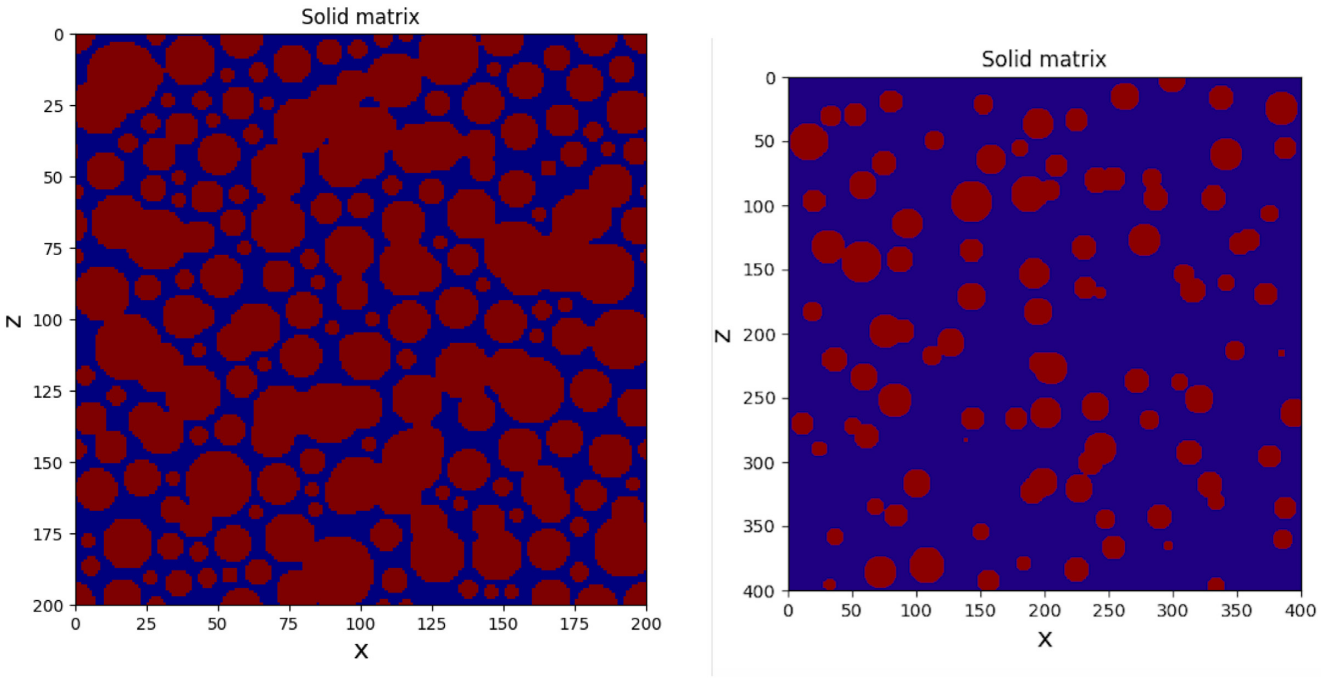

Figure 8. Two simplified rock matrices showing the solid region in red, and fluid region in blue. The matrix on the left is used for the porous flow simulation example, while the one on the right is used for the wave propagation example.

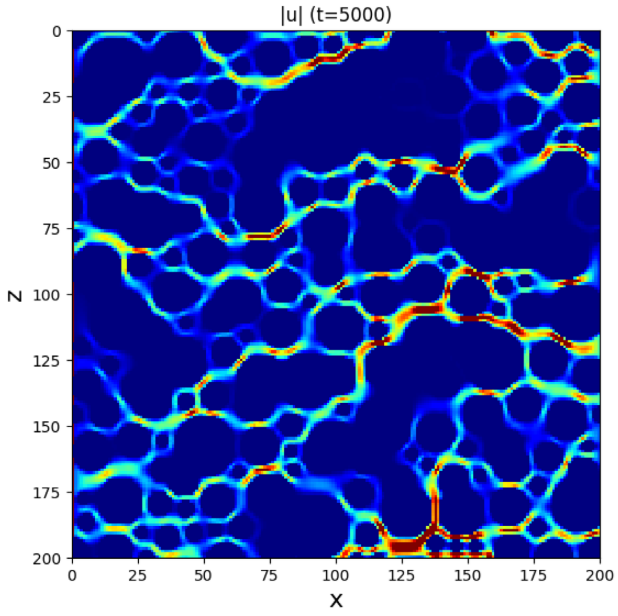

Figure 9. The fluid speed after 5000 time steps in the simulation showing the fluid pathways through the rock matrix.

We also developed an MPI parallel implementation and verified that it achieves approximately linear speedup up to 24 processors on a single node of a home computer, and more relevantly for scientific applications, that it scales linearly up to at least 1280 processors on a Beowulf cluster. We therefore recommend using Python, NumPy, JiT and mpi4py for developing scientific high performance computing software.

Scientific problems in geosciences that can be efficiently tackled with the LBM include fluid flow and wave propagation in porous media, thermochemical mantle convection, magma dynamics and volcanic eruptions. We will start off with classical 3-D mantle convection problems (e.g. Rabinowicz *et al.* 1990). Assuming that the computational time continues to scale efficiently up to 10 thousand processors on a comparable cluster to the one we used for benchmarking (10 000 Xeon processing cores, connected with 56 Gb/sec (FDR) InfiniBand 2:1), it will be possible to model a lattice size of 10^9 points ($1,000^3$) in about 3 minutes per 10 000 time steps. And if memory is sufficiently large, a lattice size of 10^{12} points ($10\,000^3$) could be calculated in about 48 hr per 10 000 time-steps. Tests on larger computing facilities are planned and will be the topic of a follow up publication.

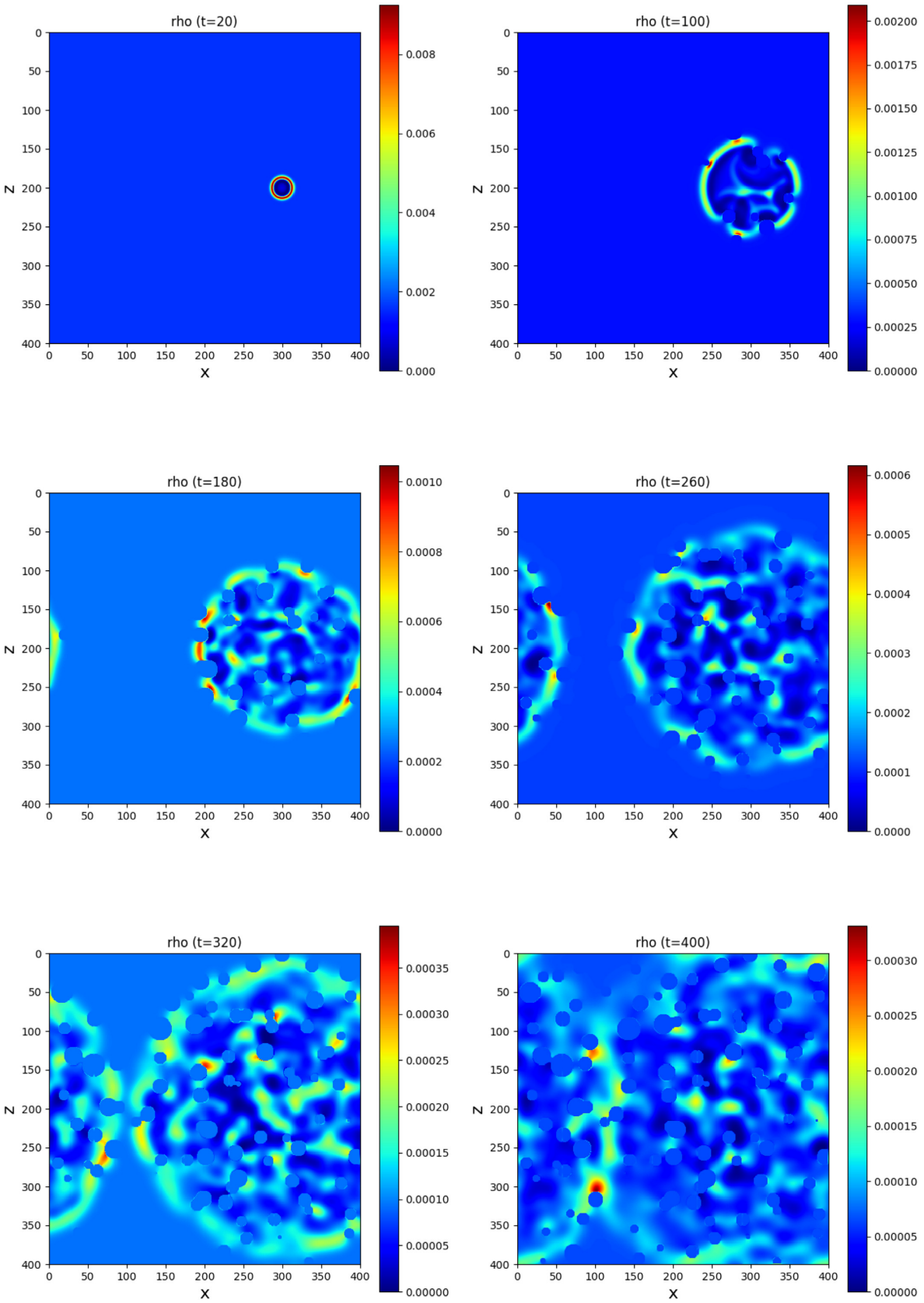

Figure 10. Snapshots showing propagation of a wave front in a fluid with solid inclusions in the model. The algorithm captures the complexities of strong scattering from the inclusions and superposition of waves.

ACKNOWLEDGEMENTS

D.A. Yuen would like to thank National Science Foundation,s geochemistry and CISE programs for support. G. Morra would like to thank the Board of Regents of Louisiana for support through the RCS project LEQSF(2014-17)-RD-A-14 and the Louisiana Optical Network Infrastructure (LONI) that provided the cluster for the test runs, through the project *loni lbm01*. D.A. Yuen and G. Morra would like to thank Matthew G. Knepley for the stimulating discussions on parallel scaling. The authors would like to thank the reviewers, C. Huber and C. Thieulot for helpful suggestions that improved the paper.

REFERENCES

Arcidiacono, S., Karlin, I., Mantzaras, J. & Frouzakis, C., 2007. Lattice Boltzmann model for the simulation of multicomponent mixtures, *Phys. Rev. E,* **76,** 046703.

Arun, S., Satheesh, A., Mohan, C., Padmanathan, P. & Santhoshkumar, D., 2017. A review on natural convection heat transfer problems by lattice Boltzmann method, *J. Chem. Pharm. Sci.,* **10**(1), 635–645.

Bartlett, S., 2017. A non-isothermal chemical Lattice Boltzmann Model incorporating thermal reaction kinetics and enthalpy phase changes, *Computation,* **5**(37), doi:10.3390/computation5030037.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S. & Smith, K., 2011. Cython: the best of both worlds, *Comput. Sci. Eng.,* **13**(2), 31–39.

Bhatnagar, P.L., Gross, E.P. & Krook, M., 1954. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems, *Phys. Rev.,* **94**(3), 511.

Chen, S. & Doolen, G.D., 1998. Lattice Boltzmann method for fluid flows, *Ann. Rev. Fluid Mech.,* **30**(1), 329–364.

Dalcin, L.D., Paz, R.R., Kler, P.A. & Cosimo, A., 2011. Parallel distributed computing using python, *Adv. Water Resour.,* **34**(9), 1124–1139.

d'Humieres, D., Ginzberg, I., Krafczyk, M., Lallemand, P. & Luo, L.S., 2002. Multiple-relaxation-time lattice Boltzmann models in 3D, *Phil. Trans. R. Soc. Lond.,* **360,** 437–451.

Di Ilio, G., Chiappini, D., Ubertini, S., Bella, G. & Succi, S., 2017. Hybrid lattice Boltzmann method on overlapping grids, *Phys. Rev. E,* **95**(1), 013309.

Feichtinger, C., Donath, S., Köstler, H.,Götz,J & Rüde, U., 2011. HPC design software for computational engineering simulations, *J. Comput. Sci.,* **2**(2), 105–112.

Frisch, U., Hasslacher, B. & Pomeau, Y., 1986. Lattice-gas automata for the Navier-Stokes equation, *Phys. Rev. Lett.,* **56**(14), 1505.

Gerya, T., 2009. *Introduction to Numerical Geodynamic Modelling*. Cambridge Univ. Press.

Groen, D., Henrich, O., Janoschek, F., Coveney, P. & Harting, J., 2011, Lattice-Boltzmann methods in fluid dynamics: turbulance and complex colloidal fluids, in *Juelich Blue Gene/P Extreme Scaling Workshop 2011,* Juelich Supercomputing Centre.

Guo, Z., Shi, B. & Zheng, C., 2002. A coupled lattice bgk model for the Boussinesq equations, *Int. J. Numer. Methods Fluids,* **39**(4), 325–342.

Guo, J., Xing, H., Zhiwei, T. & Muhlhaus, H., 2014. Lattice Boltzmann modeling and evaluation of fluid flow in heterogeneous porous media involving multiple matrix constituents, *Comput. Geosci.,* **62,** 198–207.

Guttag, J.V., 2013. *Introduction to Computation and Programming Using Python,* MIT Press.

He, X., Chen, S. & Doolen, G.D., 1998. A novel thermal model for the lattice Boltzmann method in incompressible limit, *J. Comput. Phys.,* **146**(1), 282–300.

Heuveline, V. & Latt, J., 2007. The OpenLB project: an open source object oriented implementation of lattice Boltzmann methods, *Int. J. Modern Phys. C,* **18,** 627–634.

Higuera, F.J. & Jimenez, J., 1989. Boltzmann approach to lattice gas simulations, *EPL (Europhys. Lett.),* **9,** 663 doi:10.1209/0295-5075/9/7/009.

Huang, H., Sukop, M. & Lu, X., 2015. *Multiphase Lattice Boltzmann methods: Theory and Application,* John Wiley & Sons.

Huber, C., Parmigiani, A., Chopard, B., Manga, M. & Bachmann, O., 2008. Lattice Boltzmann model for melting with natural convection, *Intl. J. Heat Fluid Flow,* **29**(5), 1469–1480.

Huber, C., Shafei, B. & Parmigiani, A., 2014. A new pore-scale model for linear and non-linear heterogeneous dissolution and precipitation, *Geochim. Cosmochim. Acta,* **124,** 109–130.

Hunter, J.D., 2007. Matplotlib: a 2D graphics environment, *Comput. Sci. Eng.,* **9**(3), 90–95.

Kang, Q., Zhang, D. & Chen, S., 2003. Simulation of dissolution and precipitation in porous media, *J. geophys. Res.: Solid Earth,* **108**(B10), 2505–2515.

Kang, Q., Lichtner, P. & Janecky, D., 2010. Lattice Boltzmann Method for reactive flows in porous media, *Adv. Appl. Math. Mech.,* **2,** 545–563.

Keehm, Y., Mukerji, T. & Nur, A., 2004. Permeability prediction from thin sections: 3D reconstruction and Lattice-Boltzmann flow simulation, *Geophys. Res. Lett.,* **31,** L04606.

Krüger, T., Kusumaatmaja, H., Kuzmin, A., Shardt, O., Silva, G. & Viggen, E.M., 2017. *The Lattice Boltzmann Method: Principles and Practice*, Springer International Publishing.

Lagrava, D., Malaspinas, O.,Latt. J. & Chopard, B., 2012. Advances in multidomain lattice Boltzmann grid refinement, *J. Comput. Phys.,* **231,** 4808–4822.

Langtangen, H.P., Barth, T.J. & Griebel, M., 2006. *Python Scripting for Computational Science,* Vol. **3,** Springer.

Lallemand, P. & Luo, L.S., 2000. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invarience, and stability, *Phys. Rev. E,* **61**(6), 6546–6562.

Lighthill, J., 1978. *Waves in Fluids,* pp. 501, Cambridge Univ. Press.

Luo, L.S. & Girimaji, S., 2002. Lattice Boltzmann model for binary mixtures, *Phys. Rev. E,* **66,** 035301.

Lutz, M., 2013. *Learning Python: Powerful Object-Oriented Programming,* O'Reilly Media, Inc.

McKinney, W., 2012. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython,* O'Reilly Media, Inc.

Morra, G., 2018. *Pythonic Geodynamics,* Springer.

Mora, P. & Yuen, D.A., 2017. Simulation of plume dynamics by the lattice Boltzmann method, *Geophys. J. Int.,* **210**(3), 1932–1937.

Mora, P. & Yuen, D.A., 2018a. Simulation of regimes of convection and plume dynamics by the thermal Lattice Boltzmann Method, *Phys. Earth planet. Inter.,* **275,** 69–79.

Mora, P. & Yuen, D.A., 2018b. Comparison of convection for Reynolds and Arrhenius temperature dependent viscosities, *Fluid Mech. Res. Int.,* **2**(3), 99–107.

Numrich, R.W., (2018). *Parallel Programming with Co-Arrays: Parallel Programming in FORTRAN*. Chapman and Hall/CRC.

Parmigiani, A., Huber, C., Bachmann, O. & Chopard, B., 2011. Pore-scale mass and reactant transport in multiphase porous media flows, *J. Fluid Mech.,* **686,** 40–76.

Qian, Y., d'Humières, D. & Lallemand, P., 1992. Lattice BGK models for Navier-Stokes equation, *EPL (Europhys. Lett.),* **17**(6), 479.

Rabinowicz, R., Ceuleneer, G., Monnereau, M. & Rosemberg, C., 1990. Three-dimensional models of mantle flow across a low-viscosity zone: implications for hotspot dynamics, *Earth planet. Sci. Lett.,* **99,** 170–184.

Schornbaum, F. & Rüde, U., 2016. Massively parallel algorithms for the lattice Boltzmann method on nonuniform grids, *SIAM J. Scient. Comput.,* **28**(2), C96–C126.

Shan, X., 1997. Simulation of rayleigh-bénard convection using a lattice Boltzmann method, *Phys. Rev. E,* **55**(3), 2780.

Shan, X. & Chen, H., 1993. Lattice boltzmann model for simulating flows with multiple phases and components, *Phys. Rev. E,* **47**(3), 1815.

Smith, K.W., 2015. *Cython: A Guide for Python Programmers,* O'Reilly Media, Inc.

Succi, S., 2018. *The Lattice Boltzmann Equation: For Complex States of Flowing Matter,* eds Succi, Sauro & Succi, S., Oxford Univ. Press.

Torquato, S., 2013. *Random Heterogeneous Materials: Microstructure and Macroscopic Properties,* Vol. **16,** Springer Science & Business Media.

Van Rossum, G. *et al.*, 2007. Python programming language, in *USENIX Annual Technical Conference,* Vol. **41,** p. 36.

Wang, J., Wang, D., Lallemand, P. & Luo, L.-S., 2013. Lattice Boltzmann simulations of thermal convective flows in two dimensions, *Comput. Math. Appl.,* **65**(2), 262–286.

Xia, M., Wang, S., Shou, H., Shan, X., Chen, H., Li, Q. & Zhang, Q., 2017. Modelling viscoacoustic wave propagation with the lattice Boltzmann method, *Scient. Rep.,* **7,** 10169.

Xie, C., Raeini, A.Q., Wang, Y., Blunt, M.J. & Wang, M., 2017. An improved pore-network model including viscous coupling effects using direct simulation by the lattice Boltzmann method, *Adv. Water Resour.,* **100,** 26–34.

Zheng, J., Ju, Y. & Wang, M., 2018. Pore-scale modeling of spontaneous imbibition behavior in a complex shale porous structure by pseudopotential lattice Boltzmann method, *J. geophys. Res.: Solid Earth,* **123,** 9586–9600.

Zhou, Q. & He, X., 1997. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model, *Phys. Fluids,* **9**(6), 1591–1598.