LLALBM Project Report

*A Lightweight, Lightspeed and Adaptable C++ implementation of the Lattice Boltzmann Method

Luca Guffanti
Cod. 10720858
luca2.guffanti@mail.polimi.it

Lorenzo Fonnesu
Cod. 10740840
lorenzo.fonnesu@mail.polimi.it

Andrea Grassi
Cod. 10741092
andrea10.grassi@mail.polimi.it

***Abstract*—In this project report we present the LLALBM C++ library, a flexible templated and header only library implementing the Lattice Boltzmann Method for Computational Fluid Dynamics, developed as a project for the Advanced Methods for Scientific Computing Course of the Master's degree in High Performance Computing Engineering at Politecnico di Milano.**

I. Introduction

Lattice Boltzmann Methods are a family of computational fluid dynamics methods that originate from lattice gas automata and have found a wide use in many fields, from quantum mechanics to image processing. These methods arise from the discretization of the Boltzmann equation, a pillar of statistical mechanics in the context of the mesoscopic kinetic theory of gases: statistical distributions of particles, which are quantities that evolve on a time scale comparable to the mean collision time between particles, are considered.

Particle behavior is modeled by a distribution function, $f(\mathbf{x}, \xi, t)$, which is a generalization of the density $\rho(\mathbf{x}, t)$ taking into account the velocity ξ vector of particles at position $\mathbf{x}$ and time t. Macroscopic quantities such as density and momentum (density) can be computed from $f(\mathbf{x}, \xi, t)$ by evaluating its moments. Considering a 3D space,

$$\rho(\mathbf{x}, t) = \int f(\mathbf{x}, \xi, t) d^3\xi, \;\; \rho(\mathbf{x}, t) u(\mathbf{x}, t) = \int \xi f(\mathbf{x}, \xi, t) d^3\xi \tag{1}$$

Finally, Boltzmann's equation describes the time evolution of the distribution function f towards its equilibrium,

$$\frac{\partial f}{\partial t} + \xi_\beta \frac{\partial f}{\partial x_\beta} + \frac{F_\beta}{\rho} \frac{\partial f}{\partial \xi_\beta} = \Omega(f) \tag{2}$$

where $\Omega(f)$, the **collision operator**, is a common notation for $\frac{df}{dt}$. Formally, $\Omega(f)$ is an integral that considers all possible outcomes of the interaction between two particles in velocity space. LB methods, however, use easier to compute operators. An example is the **BGK (Bhatnagar, Gross and Krook) Operator**, which captures the evolution of the distribution function mediated by a coefficient τ, known as **relaxation coefficient**, indicating how fast f tends to the equilibrium.

$$\Omega(f)_{BGK} = -\frac{1}{\tau}(f - f^{eq}) \tag{3}$$

LBMs revolve around **discrete-velocity distribution functions**, often called **particle populations**, $f_i(\mathbf{x}, t)$. The main difference between $f_i(\mathbf{x}, t)$ and $f(\mathbf{x}, \xi, t)$ is that velocities ξ are discretized

$$\xi \longmapsto \{\mathbf{c}_i\}$$

and constitute a set (called velocity set) that characterizes LBMs in terms of computational costs and accuracy. Additionally, points $\mathbf{x}$ on which f_i is defined (**nodes**) are distributed on a lattice with spacing Δx. Similarly, f_i is defined only at regular time steps t, with interval Δt between them. Usually, Δt and Δx are expressed in **lattice units**, that is, $\Delta t = 1$ and $\Delta x = 1$. It is possible to convert from lattice units to real world units as well as reproducing exact macroscopic behaviours if non-dimensional quantities (i.e. the Reynolds number) are equal (**law of similarity**).

Boltzmann's equation, discretized in velocity space, physical space and time becomes

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(\mathbf{x}, t) \tag{4}$$

The LBM scheme is therefore extremely simple when compared to the original equation: as time advances, nodal populations are algebraically updated according to the local action of Ω_i, the discretized collision operator of choice, (**collision step**) and transferred to neighbors (**streaming step**) as indicated by velocities $\mathbf{c}_i$. The process of streaming is shown graphically in Figure 1. As anticipated, discretized equivalents of collision operators are employed in the LBM. Different collision operators allow modeling of different phenomena, and based on their structure it's possible to analyze the contributions to the numerical stability and error of the scheme. Collision operators generally relax populations f_i towards equilibrium f_i^{eq} at different rates. f_i^{eq} being defined as

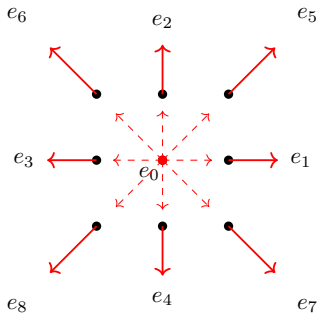

Fig. 1. Propagation in the D2Q9 velocity set, natively supported by LLALBM.

$$f_i^{eq} = w_i \rho \left(1 + \frac{\mathbf{u}\cdot\mathbf{c}_i}{c_s^2} + \frac{(\mathbf{u}\cdot\mathbf{c}_i)^2}{4c_s^2} - \frac{|\mathbf{u}|^2}{c_s^2}\right) \tag{5}$$

where w_i depends on the choice of velocities $\{\mathbf{c}_i\}$, while $\mathbf{u}$ and ρ are the local velocity vector and density, defined analogously to Equation 1 as moments of the discretized distribution function.

$$\begin{aligned} \rho(\mathbf{x},t) &= \textstyle\sum_i f_i(\mathbf{x},t) \\ \rho(\mathbf{x},t)\mathbf{u}(\mathbf{x},t) &= \textstyle\sum_i \mathbf{c}_i f_i(\mathbf{x},t) \end{aligned} \tag{6}$$

II. Structure of the Report

The report is structured in the following way: after the introduction in section I,section III provides a description of the LBM algorithm; section IV delineates LLALBM wide range of numerical methodologies; In section V we report the software design of LLALBM, its architecture and components, as well as parallelization efforts. In, section VI we show how to build a simulation from scratch. Finally, section VII shows simulation results and comparison with the suggested literature.

III. Lattice Boltzmann algorithm

In this section we describe the LBM scheme.

As already hinted in section I, the most important step of the algorithm is the populations update following the LB equation. From Equation 4, the update happens in two phases

- Populations **collide** based on the discretized collision operator of choice, $\Omega_i(\mathbf{x},t)$. The collision happens locally to each node and is an algebraic transformation, with resulting after collision populations

$$f_i^*(\mathbf{x},t) = f_i(\mathbf{x},t) + \Omega_i(\mathbf{x},t) \tag{7}$$

- After collision populations are **streamed** to neighboring nodes according to the chosen velocity set $\{\mathbf{c}_i\}$

$$f_i(\mathbf{x} + \mathbf{c}_i\Delta t, t + \Delta t) = f_i^*(\mathbf{t},t)$$

The collision-streaming step is inserted in the generic LBM iteration, completely implemented by LLALBM and described by the diagram at Figure 2.

IV. Numerical choices

Numerical choices have a great impact on the Lattice-Boltzmann method: some techniques may be simple and fast, but resulting in less precise results. Our implementation features **different solutions** to obtain **comparable results** and to **show** the great flexibility of LLALBM.

A. Velocity Sets

The choice of the velocity set is pivotal, as velocities define the neighborhood with which each node interacts. A greater number of velocities improves the ability to capture the **desired physics**, but it also increases the **numerical cost** of the algorithm, as its complexity scales with the number of velocities.

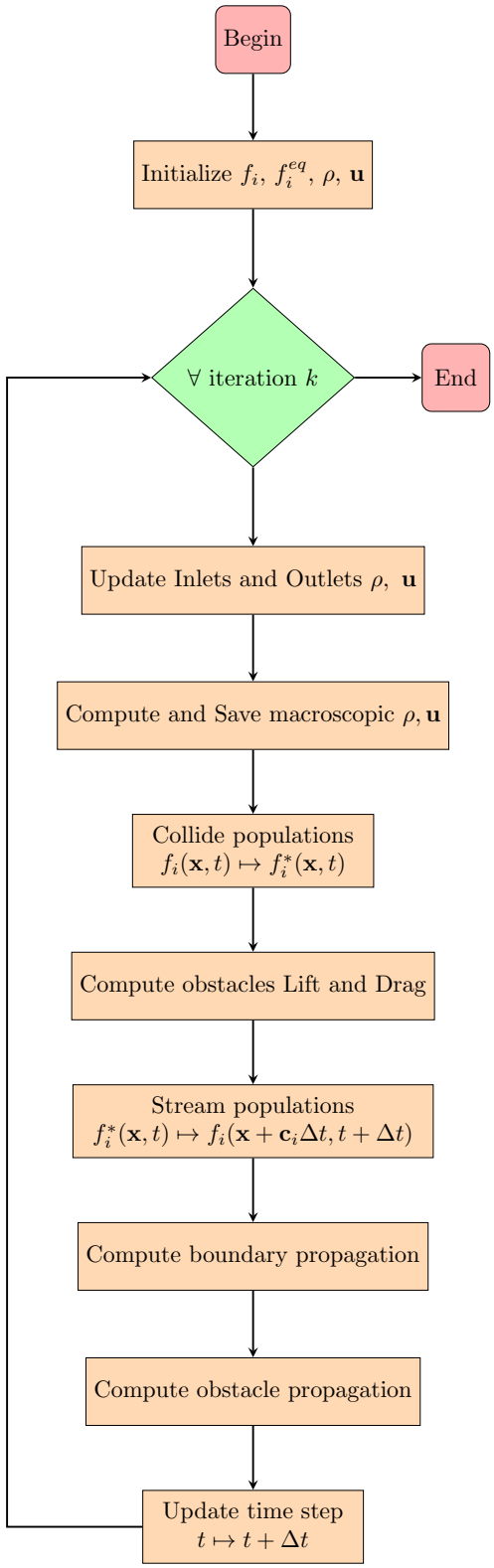

Fig. 2. Algorithm of the LB method. LLALBM exactly implements this algorithm, and allows the user to either choose the function for computation of collision and boundary/obstacle interaction, or to define a new one.

We use the notation **D*d*Q*q*** to name each velocity set by its number d of spatial dimensions and the number of discrete velocities q, the latter identifying the number of populations f_i for each node. Up to now LLALBM supports the common **D2Q9** velocity set (defined in `aliases.hpp`), which is based on the Gauss-Hermite quadrature rule, but the library can be easily expanded with other sets (the most notable are the D1Q3, D3Q15 and D3Q19 for 1D and 3D problems).

B. Collision Models

Collision models handle the interactions between fluid nodes. As already anticipated in section I, the main features of collision models is the definition of a **collision operator**, $\Omega(f)$, which captures the evolution of the distribution function at each lattice node to model microscopic particle collisions. The **collision step** is essential for redistributing particle velocities among different directions, which leads to the correct macroscopic fluid behavior. The choice of the collision operator directly affects the **accuracy**, **stability**, and **computational efficiency** of the simulation.

In the `collisions` folder we provide two of the most popular models: the **BGK (Bhatnagar-Gross-Krook)** model and the **TRT (Two-Relaxation-Time)** model, both of which offer different trade-offs between simplicity and accuracy. In our implementation, the two classes `BGKCollisionPolicy` and `TRTCollisionPolicy` expose a common interface:

- `collide` computes the collision of populations for every fluid node;
- `stream` performs the streaming to populations for all fluid nodes (equal for both BGK and TRT);
- `collide_open_boundary` computes the collision of populations at the open boundaries;
- `stream_open_boundary` performs the streaming to populations of every open boundary (equal for both BGK and TRT).

1) BGK model: The BGK model [3] is one of the simplest and most commonly used collision operators in LBM. It uses a **single relaxation rate** determined by the chosen viscosity and acts in the following way:

$$f_i(\mathbf{x} + \mathbf{c}_i\Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau}(f - f^{eq}) \qquad (8)$$

In the equation, τ, also called **relaxation time**, impacts the accuracy and stability properties of the model. Indeed, depending on the choice of τ, f_i relaxes in different ways: a big τ models a viscous fluid, while a small τ models a smooth fluid. Nonetheless, a necessary condition for stability of the scheme requires $\tau > 0.5$. Moreover, a large τ may cause an unphysical behaviour. In LLALBM τ must be initialized by the user through the static `initialize` method exposed by `BGKCollisionPolicy`.

2) TRT model: The two-relaxation-time (TRT) model [4] is a simple model that employs **two relaxation rates** in order to address the accuracy and stability issued of the BGK model. The first relaxation rate, ω^+ (or `tau_even` in LLALBM), is related to the shear viscosity while the other, ω^- (or `tau_odd`), is a free parameter. ω^+ and ω^- contribute to the stability of the scheme by acting on Λ, called **magic parameter**:

$$\Lambda = \left(\frac{1}{\omega^+\Delta t} - \frac{1}{2}\right)\left(\frac{1}{\omega^-\Delta t} - \frac{1}{2}\right). \qquad (9)$$

It can be shown that different choices of Λ lead to distinct properties: for $\Lambda = 0.25$, the method has good stability.

Implementation-wise, we find similarities with the BGK model both in the formulation

$$\begin{aligned} f_i(\mathbf{x} + \mathbf{c}_i\Delta t, t + \Delta t) = f_i(\mathbf{x}, t) \\ - \omega^+\Delta t(f_i^+ - f_i^{eq+}) \\ - \omega^-\Delta t(f_i^- - f_i^{eq-}). \end{aligned} \qquad (10)$$

and in the necessary initialization of the two parameters, required by LLALBM, before any actual computation can be done.

Finally, the TRT model is particularly useful when accurate modeling of flow behavior near boundaries is critical, when different directions exhibit different physical properties or when multi-scale phenomena are present, thanks to the coupled behaviour of the two relaxation parameters.

C. Boundary Conditions

In LLALBM, boundary conditions are enforced on different type of nodes, defined in `aliases.hpp`:

- `Fluid` nodes compose the bulk of the fluid.
- `Solid` nodes are those not in contact with fluid nodes.
- `Boundary` nodes link fluid and solid nodes and require special dynamical rules.
- `Inlet & Outlet` nodes are open boundary nodes, meaning that they behave like fluid ones, but their macroscopic quantities are forced at each iteration.
- `Obstacle` nodes are special boundary nodes that constitute obstacles in the system.

In this context, boundary conditions are a crucial component of Lattice Boltzmann Methods, as they define how particles **interact** with the physical boundaries of the domain. Accurate boundary conditions are essential for ensuring that the simulated fluid behavior matches real-world phenomena. In LLALBM, boundary condition methods are implemented in the `boundaries` folder and each of them exposes the `update_boundaries` method, which computes the collision between fluid nodes and boundaries/obstacles. More specifically, we implemented three methods: **Bounce-Back**, **Partially saturated Bounce-Back** and **Zou-He**, which are described in the sections below.

1) Bounce-Back: This method, implemented in `BounceBackPolicy.hpp`, is the simplest approach to deal with boundary and obstacle nodes, and is particularly effective for simulating no-slip boundaries, i.e. boundaries where the fluid velocity is zero.

Its effects on fluid nodes can be easily calculated as the exact reflection of populations hitting a rigid wall during

propagation from where they originally came from. Mathematically, this can be expressed as:

$$f_{\bar{i}}(\mathbf{x}, t + \Delta t) = f_i(\mathbf{x}, t). \tag{11}$$

Other then its **simplicity** (which makes BB useful for 3D simulations too), this method is also a **stable** numerical scheme, but it **loses** accuracy when dealing with curved obstacles.

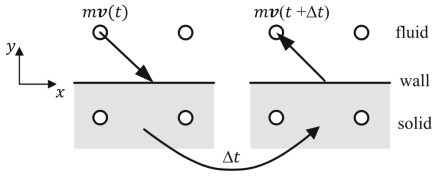

Fig. 3. Sketch of a moving particle hitting a rigid wall.

2) Partially saturated Bounce-Back: The Partially saturated Bounce back method [2], known as continuous bounce-back, aims at resolving bounce-back issues regarding its approximation with obstacles. For this reason, in LLALBM, we utilize it when dealing with circles or more complex obstacles.

The bounce-back condition is modified to account for the partial saturation of the boundary, with the reflection of the distribution function depending on the degree of wetting of each node, allowing the simulation to more accurately represent the interaction between the fluid and the boundary. Obstacle nodes can be pure or mixed, and this distinction is implemented by introducing the `b` attribute in `ObstaclePoint`.

LLALBM requires the user to call `compute_obstacle_weight` (implemented in `Lattice.hpp`) before any actual computation is performed, in order to precompute the `b` parameter for each obstacle node.

`b` is then used at each iteration to evaluate the reflected populations:

$$\begin{aligned} f_i(\mathbf{x} + \mathbf{c}_i\Delta t, t + \Delta t) = f_i(\mathbf{x}, t) \\ - (1 - b)\left(\frac{f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)}{\tau}\right) \\ - b((f_{\bar{i}}(\mathbf{x}, t) - f_{\bar{i}}^{eq}(\rho, \mathbf{u})) \\ - (f_i(\mathbf{x}, t) - f_i^{eq}(\rho, \mathbf{u}_s))), \end{aligned} \tag{12}$$

where the term multiplied by $1 - b$ models the fluid presence in the obstacle node.

3) Zou-He: The Zou-He boundary condition [5](or **NEBB**, Non-Equilibrium-Bounce-Back) is a popular **third-order accurate** method used in LBM to impose velocity or pressure at boundaries. More specifically, it is particularly effective with **open boundaries** where non-zero velocities or pressures are allowed, this being an essential aspect to simulate realistic flow scenarios.

Zou-He revolves around the concept of **missing populations**, which are those populations belonging to the boundary node that are "in contact" with fluid nodes (in a top wall the missing populations are f_4, f_7 and f_8) and that constitute problem there is no neighboring node that can propagate them to the boundary, as such nodes would either be solid or outside the domain. Zou and He solve this problem by using the enforced macroscopic quantities (density and velocity) to evaluate unknown populations, in a three step approach (here shown for top walls).

1) Identification of the missing populations.
2) Determination of the wall density independent of the unknown populations.

$$\rho = f_0 + f_1 + f_3 + 2(f_2 + f_5 + f_6). \tag{13}$$

3) Computation of unknown populations by enforcing macroscopic boundary conditions and non-equilibrium distributions.

$$f_4 = f_2 - \frac{2}{3}\rho u_y, \tag{14}$$

$$f_7 = f_5 - \frac{1}{6}\rho u_y - \frac{1}{2}\rho u_x + \frac{1}{2}(f_1 - f_3), \tag{15}$$

$$f_8 = f_6 - \frac{1}{6}\rho u_y + \frac{1}{2}\rho u_x - \frac{1}{2}(f_1 - f_3). \tag{16}$$

In this way, the NEBB method modifies **only the missing populations**.

D. Drag & Lift

In fluid dynamics, drag and lift are the two primary aerodynamic forces that act on an object moving through a fluid, such as air or water. These forces are critical in the design and analysis of various systems, as they influence the stability, efficiency, and performance of those systems. In particular:

- `Drag`: force that opposes the motion of an object through a fluid, in the direction of motion;
- `Lift`: force that acts perpendicular to the direction of motion.

Both forces are influenced by the shape and size the object, and the speed of the flow around it, as well as the properties of the fluid, such as its density and viscosity.

Drag and lift are computed in `FlowAnalyzer.hpp`, where the `momentum exchange method` is employed. Each "wet" obstacle node constitutes a unit of surface and is analyzed individually after particle collisions have been computed: a first order bounce back is simulated and momentum is evaluated as the total populations moving towards the node or in the opposite direction. In the $D2Q9$, drag is computed with populations $f_1, f_3, f_5, f_6, f_7, f_8$, while lift is computed with populations $f_2, f_4, f_5, f_6, f_7, f_8$. In this setting, populations evolving in an oblique direction are not considered as is, but are rescaled by a factor $\sqrt{2}/2$, to actually consider the projection of the momentum vector along the cardinal direction of interest.

V. LLALBM Software Architecture

LLALBM is delivered as a header only library, powered by `Eigen`, where flexibility is the key. The software architecture and infrastructure enable the user to seamlessly introduce functionalities or use pre-existing ones, as the large API allows fine control of simulations.

A. Core Policy-Based Design and Lattice Object

In order for LLALBM to be flexible and easily user expandable, the majority of the library is policy based. The use of policies and templates requires effort, but allows compile time policy resolution, consistency checks, and avoidance of polymorphic approaches which can degrade performances. Policies, encapsulated by C++ classes, are of four types.

- **Collision policies** regard collision operators.
- **Boundary policies** regard fluid-boundary or fluid-obstacle interaction.
- **Initialization policies** regard the initialization of lattice nodes, inlets and outlets.
- **Equilibrium policies** regard the computation of equilibrium populations.

Such policies can be used to choose the numerical methods employed by the simulation thanks to standardized interfaces that expose static methods, among which those used for initialization purposes. Policy typenames are collected by the `LatticeConfiguration` static object as type aliases, and are passed to `LatticeConfiguration` as template parameters. `LatticeConfiguration` has eight template parameters: the physical dimensions of the problem, and the seven available LLALBM policy uses.

- `CollisionPolicy`, to specify the collision operator.
- `WallPolicy`, to specify the fluid-boundary interaction.
- `ObstaclePolicy`, to specify the fluid-obstacle interaction.
- `InletPolicy`, to specify the inlet behavior.
- `OutletPolicy`, to specify the outlet behavior.
- `InitializationPolicy`, to specify how nodes should be initialized and updated.
- `EquilibriumPolicy`, to specify how equilibrium populations should be computed.

Policy consistency is checked at compile-time with LLALBM `C++ concepts`, defined in `PolicyTypes.hpp`.

The remaining infrastructure accesses policies from aliases provided by `LatticeConfiguration`. The most important class that benefits from compile time policy resolution is the core of the system: the `Lattice` class.

`Lattice` objects are templated and instantiated by the user, who must pass `ParallelizationPolicy` typename as a template parameter. `ParallelizationPolicy`, that will be described later, requires a `LatticeConfiguration` typename as a template parameter, through which it is capable of accessing policies.

`Lattice` constitutes the main system that governs the entire LBM simulation: when templating is resolved, `Lattice` extracts the number of dimensions from the `ParallelizationPolicy` parameter and constructs the necessary data structures with domain information provided by the user. The most important `Lattice` members are

- `Eigen::Tensor` objects that house populations, after collision populations, and the global density and velocity fields.
- `std::vector` objects that hold the coordinates of fluid nodes, inlet nodes, outlet nodes, boundary nodes and obstacle nodes. **Fluid nodes** coordinates are stored as `Point<dim>` objects, while **inlet**, **outlet** and **boundary nodes** are stored as `BoundaryPoint<dim>`, which are objects that contain coordinates and the type of boundary found at the node, the latter implemented with an enumeration. Finally, **obstacle nodes** data is stored as `ObstaclePoint<dim>` objects which contain coordinates and a binary mask indicating in which directions (dependent to the chosen velocity set) fluid nodes are found.
- A `Logger` object used for logging purposes.
- A **std::shared_ptr** to a `FlowAnalyzer<dim>` object, used for intra-simulation evaluation of the lift and drag forces on obstacles. The `FlowAnalyzer` object is described in paragraph V-B1d

`Lattice` objects support reinitialization to allow the definition multiple simulations within the same object, and printing of the most relevant information. Still, the most important method exposed by `Lattice` is `perform_lbm()`. This method, which accepts the total simulation time, the time step and information about saving as parameters, is used to compute the simulation. Aside from secondary chores (setting up the output directory and similar), `perform_lbm()` implements exactly the algorithm of which at Figure 2, with additional preprocessor macros that enable verbose logging, timing, and numerical stability checks on tensors with the `std::is_nan()` function. Numerical tasks, aside from the computation of the Lift and Drag which is done separately, are offloaded to the `ParallelizationPolicy` object that is passed as a template parameter and called via static methods invocations. The `ParallelizationPolicy` object accepts a `LatticeConfiguration` as a template parameter and acts as an intermediary between `Lattice` and policies to enable their invocation while providing scopes into which initialization, cleanup, and eventual parallelization-dependent procedures can be called without interacting with `Lattice` itself. `ParallelizationPolicy` is therefore an evolution of the *Delegate* design pattern. However, `ParallelizationPolicy` instances should only concern fine-grain parallelization, such as that provided by `OpenMP` and `CUDA`. LLALBM considers domain distribution (like that obtained with MPI) as being one layer above, requiring a specific implementation of the `Lattice` object, to merge both parallelization levels seamlessly.

B. Other Systems and Components

The previous section presents the core of the LLALBM library. Nonetheless, the entire infrastructure cooperates with `Lattice` to allow an easy setup of simulations and post simulation analyses.

1) Nodes Initialization: Populations and velocities must be initialized at the beginning of a simulation, and inlets and outlets velocities or densities must be updated at each iteration of the method to take into account the presence of forcing

terms. To have maximum flexibility, initialization systems are templated with respect to the number of dimensions and grouped in the **InitializationPolicy** family, and must expose a set of functions that enforce open boundary conditions and define inlet and outlet nodes (**update_nodes()** and **attach_nodes()**).

Updates are computed by evaluating user defined arrays of **std::function<double(double, BoundaryPoint<dim>)>** functions that compute the value (or vector) of the field of interest depending on the coordinates of each considered open boundary point at any instant of the simulation. Such functions must be attached to the chosen initializer before the simulation starts, via a call to the **Initializer<dim>::attach_update_functions()** static method.

a) Lattice Reading and Writing: LLALBM provides two systems for I/O purposes. **LatticeReader**, generic with respect to the dimensions of the problem, is built to read a **.txt** file with a standardized format:

```
dimensions
width height depth ...
#Fluid #Solid #Boundary
#Inlet #Outlet #Obstacles on one line
y1 x1 z1 ... type of node 1
y2 x2 z2 ... type of node 2
y3 x3 z3 ... type of node 3
y4 x4 z4 ... type of node 4
```

Points should be indexed as if they were in a matrix with origin in the top-left corner.

LatticeReader is called by default when the **Lattice** object is constructed with a path to a file as a constructor parameter and is capable of generating computational domains for any wanted number of dimensions. To do that **LatticeReader** indexes elements of a generic **n**-dimensional tensors, and this is done with the **TensorLoopAccessor** metafunction. Simulations produce data that can be stored to disks, and to do that LLALBM exposes the **LatticeWriter** objects that stores the fields of interest to the **results/** directory.

b) In-code Lattice Generation: The computational domain can be directly generated in-code, without using lattice reading functions and polluting the workspace with input files. The core of such system is the templated **ConstructionInfo<dim>** object, which is used to introduce domain construction information, internally represented as **std::set** objects of **BoundaryPoint<dim>** or **ObstaclePoint<dim>**. Currently, **ConstructionInfo<dim>** exposes methods that construct the domain and add nodes of different types,

- **attach_domain_dimensions()** to define the extension of the domain.
- **add_nodes_interval()** to add a segment of nodes of a chosen type, by specifying the origin, the end point and the type.
- **add_perimeter_nodes()** to distribute nodes of a chosen type, passed as a parameter, on the perimeter.

and methods that directly construct obstacle nodes.

- **add_obstacle_hyper_rectangle()** to build an hyper rectangle starting from the uppermost and leftmost vertex and the extensions along each direction.
- **add_obstacle_hyper_square()** to build an hyper square.
- **add_obstacle_hyper_sphere()** to build an hyper sphere starting from the origin and the radius.
- **add_obstacle_random_spheres()** to build random spheres uniformly generated with the **std::mt19937** random number engine, to hint at the behaviour of simulations in the case of flow in porous media.
- **read_obstacle_from_file()** to read an obstacle file containing coordinates of obstacle nodes in order to introduce complex domains.

These methods either directly enumerate the nodes or generate a **dim**-dimensional grid of from which nodes are then filtered out. Information present in **ConstructionInfo** is then passed to the **build_lattice()** procedure that computes fluid nodes coordinates, identifies the boundary type, strips the obstacles list of non wet nodes and computes the bitmask for each obstacle node. **build_lattice()** **reinitializes** an already-existing **Lattice** object passed by reference by injecting the new lists of nodes and domain geometry. It is important to note that the procedures for characterizing obstacles and boundaries depend on the problem's dimensionality and the chosen velocity set. These procedures require calling functions whose generic definitions are currently empty, except for the D2Q9 velocity set, which is the only one completely supported so far.

c) Flow Analysis: LLALBM contains an easily extendable flow analysis system centered around the **FlowAnalyzer<dim>** class, which supports computation of lift and drag. Such class is templated with respect to dimensions and the generic implementation left empty. Specialization is required due to the strong dependency on the chosen problem dimensions and velocity set, and up to now **FlowAnalyzer** is specialized for the **D2Q9** velocity set. **FlowAnalyzer<dim>** provides numerous methods that can be used to manage the obstacles of interests.

- Users can insert single nodes or **std::vector** vectors of nodes with the **add_point()** and **add_point_vec()** methods.
- Users can let FlowAnalyzer automatically compute all obstacle nodes linked to an origin obstacle node. This happens recursively, and it is kick-started with a call to the **recognize_nearby_obstacle()** method, that then calls the internal recursive method **compute_nearby_obstacle()**.

In any case, **FlowAnalyzer** exposes the **match_wet_nodes()** method, which can be used to filter all non-wet obstacle nodes that may have been

wrongfully considered to avoid useless computations. **print_considered_points()** can instead be used to log all the registered points to **std::cout** or to a **.csv** file for domain analysis and visualization purposes.

compute_flow_properties() is the core of **FlowAnalyzer**: it implements the computation of lift and drag by simulating a first order bounce back between fluid particles and obstacles, and considering the momentum exchange.

Independent of the chosen specialization of **FlowAnalyzer**, LLALBM supports its use in two different contexts.

1) After a simulation is completed. In this case **FlowAnalyzer** must be instantiated after the simulation and correctly initialized by passing obstacle points as described. A call to the method **compute_flow_properties()** computes lift and drag that are returned inside a **std::pair**.
2) At regular intervals during a simulation. In this case **FlowAnalyzer** is considered do be owned both by the user and by the **Lattice** object that manages the simulation. To accommodate this semantic, **FlowAnalyzer** extends **std::enable_shared_from_this** and implements **std::shared_ptr<FlowAnalyzer<dim>> create()**, so that **Lattice** can safely get a shared pointer to the object by calling the **FlowAnalyzer** member function **shared_from_this()** on the shared pointer passed by the user.

FlowAnalyzer can be set to save analysis data by calling **set_should_save(true)** method, in which case data is stored in the **flow_analysis/** directory in **.csv** format. Nonetheless, global data is accessible at runtime with appropriate getters, while single node contributions are deleted after the computation.

d) Other utilities: Among the already presented systems, LLALBM provides two optional utilities that aid development.

- Logging system. The **Logger** object is a wrapper around an output stream that is passed as a constructor parameter together with name of the logging system. **Logger** provides three possible levels of logging: **info, warning, error**, and the format of logs is the following

```
[%NAME%-%LEVEL%] %MESSAGE%
```

- Multi Dimensional Loop. The `MultiDimensionalLoop` utility is essentially a **metafunction** - using **SFINAE** principles - to allow compile-time definition of for loops that iterate through all the coordinates of a multi-dimensional grid, with or without offset. **MultiDimensionalLoop** is templated with respect to
  - **typename ContainerObject**: a container type that implements the **push_back()** method.
  - **typename DataObject**: the data type contained by **ContainerObject**. **DataObject** must expose a constructor that accepts either **std::array<Eigen::Index, dims>** or **std::initializer_list<Eigen::Index>** objects.
  - **std::size_t dims**: the number of dimensions
  - **std::size_t current_dim**: the current dimension, starting from 0.

  Calls to static **MultiDimensionalLoop** methods are solved in terms of number of dimensions at compile time. In this way the runtime uses different partial specializations of **MultiDimensionalLoop**, dependent on the **current_dim** template parameter, to perform recursive calls to visit higher dimensions or to close the recursion stack when **current_dim==dims**. Grid points are therefore generated in a depth-first fashion and pushed back in a **ContainerObject** instance passed by reference to the calls.
  MultiDimensionalLoop is used especially in nodes generation, recognition, and lattice construction. Nonetheless, it is available to the entire code-base.

C. Parallelization

As already stated, every **Parallelization policy** is an intermediary between **Lattice** and the numerical policies contained in a LatticeConfiguration. Consistency checks are necessary to verify the correctness of the policies supplied to **LatticeConfiguration**. Such verification is done via **C++ concepts**, introduced in **C++ 20**. Concepts are named boolean predicates on template parameters that encode conditions checked at compile time: LLALBM uses them to ensure that every policy in **LatticeConfiguration** is associated to the same parallelization policy (done with a tag), throwing easy-to-understand errors at compile time if inconsistencies are spotted.

LLALBM is easily extendable with new parallelization polices by creating a new parallelization policy class together with a respective tag.

Parallelization techniques already supported by LLALBM are the following:

- **OMP**: Open Multi-Processing is an API that introduces special preprocessor **#pragma omp** directives to allow fine parallelization with multithreading. OMP is both relatively easy to implement and effective since LBM iterations contain many independent for loops: having many active threads notably speeds up the execution of the algorithm, although thread creation and synchronization overhead can become wasteful. In LLALBM threads are spawned every time a parallel policy is invoked, simulations must be benchmarked to see which configuration is the best.
- **STD::EXECUTION**: **C++** execution policies are introduced with **C++ 17** and are used to specify the approach to executing parallel algorithms belonging to the **C++ Standard Template Library**. For loops are substituted with calls to **algorithm** functions, which are

overloaded to accept an execution policy as a first parameter. LLALBM has no control over whether the algorithm is actually parallelized as an eventual parallelization is handled internally: the actual number and type of used threads depends on the implementation of the standard library present in the system.

- **OpenACC**: Open Accelerators is a programming model adopted by LLALBM thanks to its simplicity and portability. Similarly to OMP, OpenACC provides a high-level, directive-based approach to parallelization with compiler directives (**`#pragma acc`**) to offload computations to accelerators. LLALBM employs OpenACC to pave the way for a future GPU / FPGA parallelizations of the algorithm, that will be easily introduced in the already present system thanks to the simple **`pragma`** preprocessor directives.

D. Additional Python Scripts

`Python` scripts supplement the LLALBM with functionalities that require fast and flexible management of data, as they rely on well-known libraries for image processing (**`PIL`**), numerical processing (**`NumPY`**) and data analysis(**`Pandas`** and **`MatPlotLib`**). Scripts are stored in the **`scripts/`** directory and are the following

- **`plot_lift_drag.py`** is used to build lift and drag plots from data stored in **`.csv`** files by **`FlowAnalyzer`**. This script expects the directory into which flow analysis data is stored as a command line parameter.
- **`plot_obstacle_layout.py`** is used to plot the obstacle nodes layout that is considered by **`FlowAnalyzer`** for computation of lift and drag. This script expects the **`.csv`** file into which coordinates of obstacle nodes are stored as a command line parameter.
- **`produce_video.py`** is used to generate an animation of the simulation. It requires the directory into which simulation data is stored in **`.txt`** files as well as the field of interest (LLALBM only prints the velocity field, there described as **`u`**). In the case of a velocity field, if the user is interested in animating only the velocity norm, the **`norm`** command line parameter is expected.
- **`translate.py`** is used to build input fields and lattices. It expects an image file (**`.png`**) which, based on the color of pixels, is converted to a representation that is comprehensible by LLALBM. Up to now LLALBM only supports lattice insertion from file.
- **`build_obstacle_file.py`** is used to traduce a black-and-white **`.png`** image depicting an obstacle into a set of node coordinates that can be then inserted in a domain. This script expects the path to the image as a command line parameter.

VI. LLALBM SIMULATION WORKFLOW

One of LLALBM's main objectives is making the process of building a simulation easy and accessible. The workflow is centered around different phases, and better shown in Figure 21 as well as in the various examples we propose.

1) **Defining LLALBM macros.** LLALBM allows definition of preprocessor macros that activate or deactivate certain sections of the code
   - **`LLALBM_DEBUG`** for numerical stability checks on velocity tensors,
   - **`LLALBM_VERBOSE`** for verbose logging.
   - **`TIMER`** for timing of LB iterations.
2) **Choosing a lattice configuration and parallelization.** The lattice configuration and parallelization policies are chosen by defining appropriate type aliases.
3) **Constructing the lattice and initializing policies.** There are two ways of constructing the computational domain. If lattice information is stored in a **`.txt`** with a format compliant to the one supported by LLALBM, it is enough to construct an instance of the **`Lattice`** object by passing the path to the input file as a parameter, together with the number of velocities in the velocity set and the **`std::ostream&`** that is used for logging purposes. In order to build a domain by exploiting the in-code generation, one should first instantiate an empty **`Lattice`** object, and a **`ConstructionInfo`** object that would be populated with lattice information. Subsequently, the **`build_lattice`** method should be called to reinitialize the lattice.
   Now, a subset of policies should be initialized independent of the approach for generating the domain, and initialization is done by calling the **`initialize`** static method exposed by the policies.
4) **Running the simulation** The simulation can be run as the setup phase is concluded: one should call the **`perform_lbm()`** method exposed by the **`Lattice`** object with the appropriate parameters.
   If however one was interested in the computation of lift and drag during the simulation, for instance to see how lift and drag vary as the inlet flow profile changes over time, the **`FlowAnalyzer`** object should be instantiated and initialized through a **`std::shared_ptr`** and passed to the **`Lattice`** object that will manage it during the simulation.
5) **Analysing the results of a simulation.** Obviously, LLALBM allows lift and drag analysis after a simulation is completed: the **`FlowAnalyzer`** should be instantiated on the stack, initialized, and called to compute the properties of the flow by passing the population tensor extracted from the **`Lattice`** object.
6) **Using Python scripts to produce plots and animations.** Finally, after execution, **`Python`** scripts can be used to plot results and animate the simulation, by calling the following.

VII. RESULTS AND SIMULATIONS

LLALBM allows the user to construct a wide variety of simulations, and is distributed with ten examples that showcase the library capabilities.

- **`1_LidDrivenCavity`**, **`1_LidDrivenCavity129`**. A 100×100 and a 129×129 Lid driven cavities are

explored as first simulations, with successful results. As shown by the velocity norm at Figure 4, the simulation is compliant with results from the literature. Additionally,

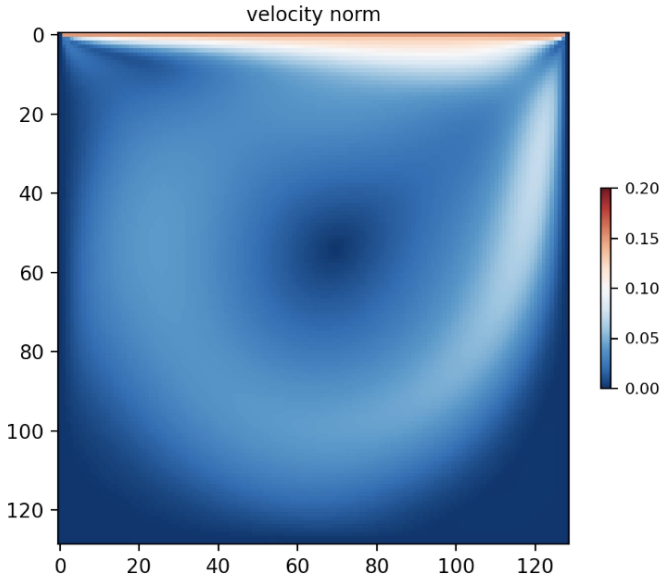

Fig. 4. Lid driven cavity simulation. Reynolds is estimated to be around 1000.

the Lid driven cavity has been compared against [1], as shown by Figure 5, where velocities in the lines crossing the geometrical center of the domain are plotted.

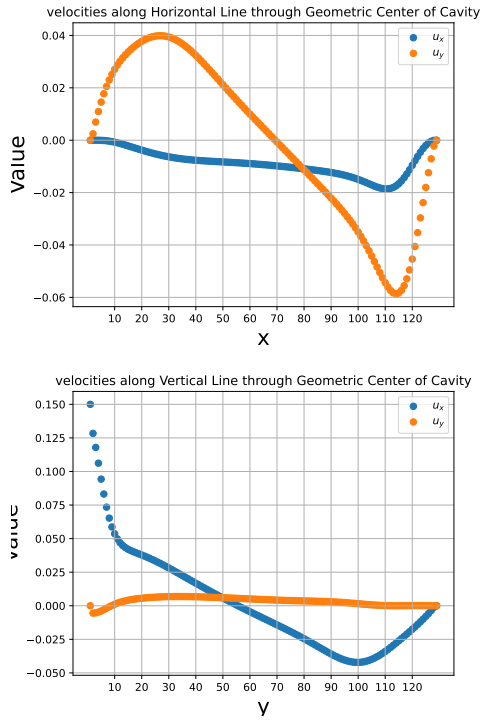

Fig. 5. Horizontal and vertical velocities in the horizontal and vertical lines through the geometric center.

- **`2_Parabolic`**. In this example a parabolic flow is simulated in a 200×50 horizontal channel, with a maximum horizontal velocity $v = 0.2$ imposed on the inlet node in the middle of the boundary. The velocity norm is showed in Figure 6 and it is interesting to notice how

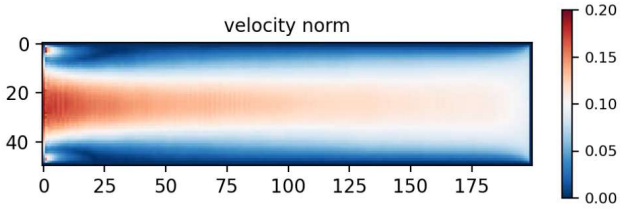

Fig. 6. Parabolic flow profile in a channel.

some artifacts begin to appear due to subtle numerical instabilities.
- **`3_Vortex`**. This example shows two use cases in which a flow around a cylinder is simulated. The flow profile is uniform, and reaches through a sigmoid a maximum velocity of 0.2. Noticeably, the cylinder is placed asymmetrically in the domain, to stimulate the generation of Kármá vortices, more present in a less viscous fluid. Results are shown in Figure 7.

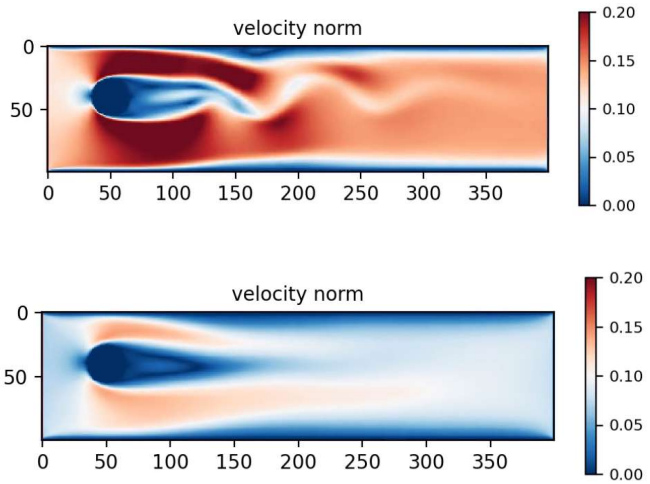

Fig. 7. Above: vortices are present when the fluid is less viscous and higher velocities and turbolences are achieved. Below: a high relaxation parameter makes the simulation less chaotic, even though tail effects are somewhat visible.

- **`4_ComplexGeometry`**. In this example we showcase the ease of generating multiple obstacles with the in-code lattice generation infrastructure. We do not provide images here, but the related **`gif`** is present in the **`README`** of the repository.
- **`5_ RandomGeneration`** LLALBM is capable of using random number generation to distribute spherical ob-

stacles in a computational domain. Randomization affects both radius and center coordinates of each sphere. An example is shown in Figure 8, where a 700×100 domain is populated and subjected to a uniform inlet with velocity increasing from 0 to 0.2 following a sigmoid function.

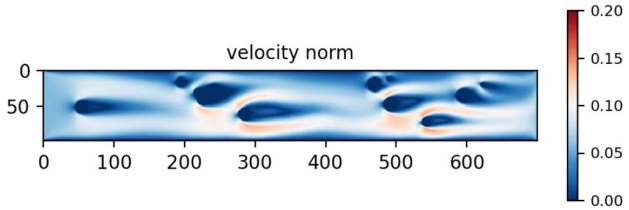

Fig. 8. Flow around a random domain.

- **`6_FlowAnalysis`**. As showcased in the previous sections, LLALBM offers an infrastructure for computing Lift and Drag around an obstacle. Example 6 shows how to introduce this system in already existing simulations, expanding on the results of Example 3. However, to avoid instabilities due to the presence of Kármán vortices, the cylindrical obstacle is placed in the middle of the 400×100 domain and a tail is introduced. The flow around the obstacle is shown in Figure 9 while

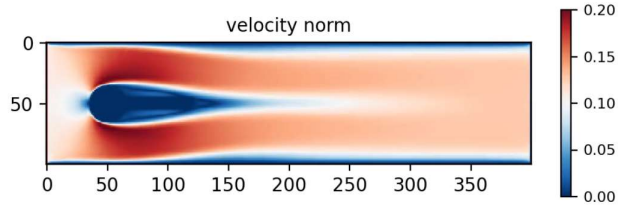

Fig. 9. Flow around a cylindrical obstacle with a tail, in order to avoid trailing vortices.

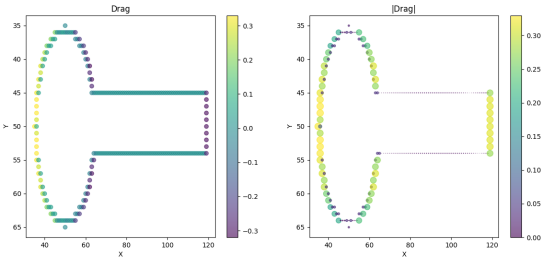

Fig. 10. Drag contribution of each obstacle node, also shown in magnitude.

results from the flow analysis are presented in Figure 10, Figure 11 and Figure 12. LLALBM stores individual node contributions to lift and drag in order to produce specific plots, and global values of lift and drag that allow the definition of plots showcasing the evolution of these quantities over time, as iterations of the method progress. The lift force is positive when it points up, and the drag force is positive when pointing right.

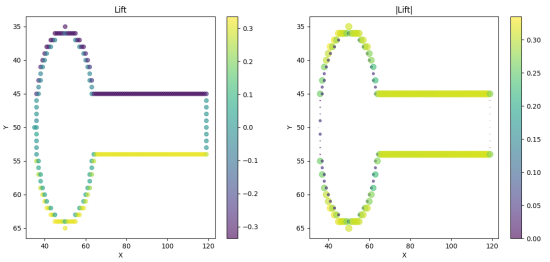

Fig. 11. Lift contribution of each obstacle node, also shown in magnitude.

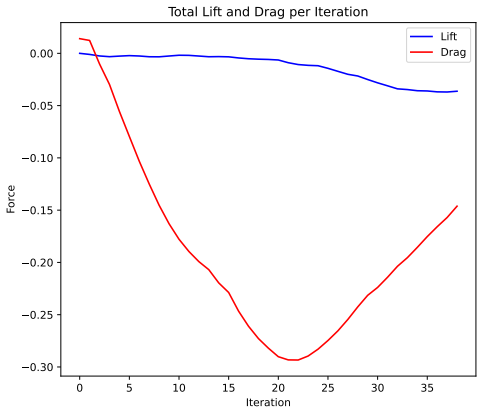

Fig. 12. Lift and drag acting on the cylinder of Example 6.

- **`7_FlatBottomFoil`** This example showcases the use of the lattice generation infrastructure coupled with the obstacle reading functionality provided by LLALBM, as a flat bottom aerofoil is simulated. The 600×200 domain has an inlet that imposes an horizontal velocity $v = 0.2$. Given the configuration of the simulation with a high relaxation parameter we expect a laminar flow around the obstacle, with a stable lift once the velocity becomes stable. These expectations are confirmed by Figure 13 and 14.

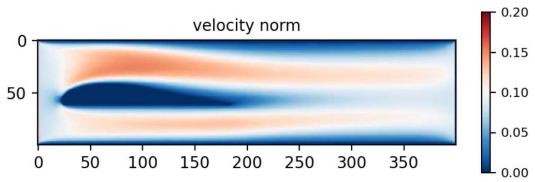

Fig. 13. Flow around a flat bottom aerofoil in a laminar context.

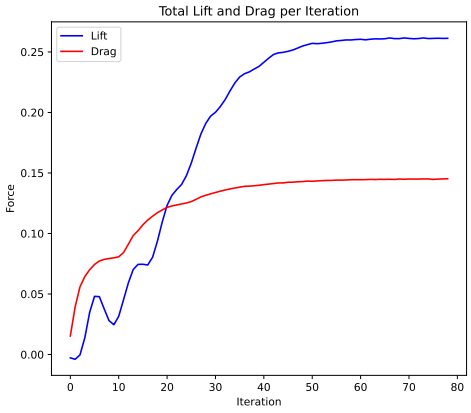

Fig. 14. Total lift and drag around the flat bottom aerofoil as time progresses. The foil produces positive lift.

- **`8_N_NACAProfile`**. Examples **`8_1_NACAProfileFlight`** and **`8_2_NACAProfileTakeOff`** show flow around a NACA aerofoil in the contexts of flight and takeoff, considering both laminar and turbulent regimes. These examples are obtained by generating an obstacle file introduced as in Example **`7_FlatBottomFoil`**. Example **`8_1_NACAProfileFlight`** uses a 400×100 domain with a horizontal inlet flow on the left reaching a speed of 0.2. Laminar flow happens with $\tau^+ = 0.9$ in with a **`TRT`** obstacle policy, necessary for stability. Lower relaxation rates produce slight turbulence. Figure 15 shows the flow around the foil, while Figure 16 presents the lift and drag analysis. Fluctuations in the lift are notable most probably due to vortices in the tail of the foil. Yet, it is noticeable how the NACA foil behaves

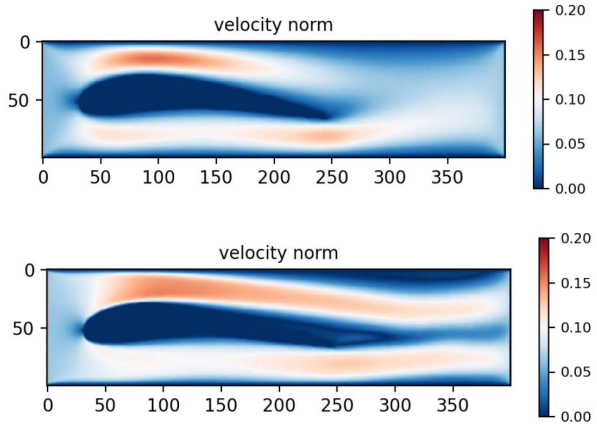

Fig. 15. Flow around a NACA foil in a laminar regime (above) and more turbulent regime (below) during flight.

better in laminar flow conditions, with the maximum lift being almost twice as the maximum lift in the turbulent context. Example **`8_2_NACAProfileTakeOff`** shows the flow around a NACA aerofoil during the

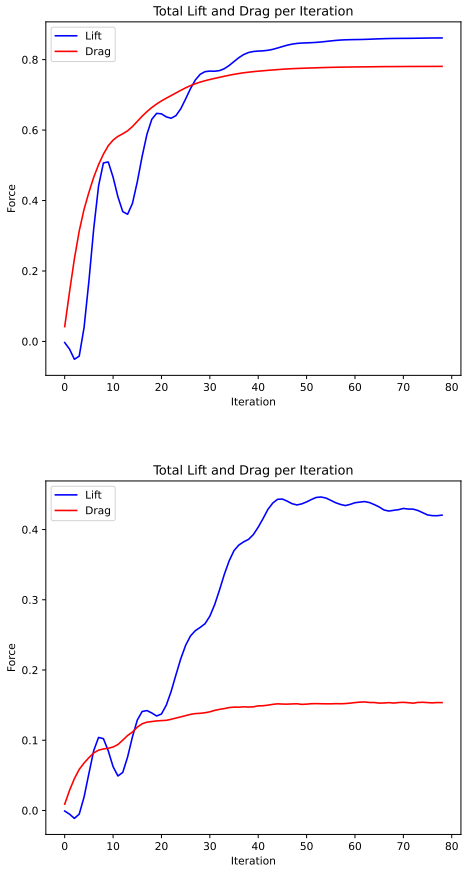

Fig. 16. Flow analysis for a NACA foil in a laminar regime (above) and more turbulent regime (below) during flight.

takeoff phase, in laminar and turbulent flows. Simulating takeoff required precise management of the inlet velocity functions, made easy thanks to LLALBM initialization policies: in order to model the change in angle of attack due to the plane climb inlets were configured to impose a vertical velocity after a first horizontal displacement phase simulating taxiing. The mesh was adapted to take into account the presence of flaps. Figure 17 shows the velocity norm in the laminar and turbulent cases, while 18 shows the lift and drag analysis. It is noticeable how the NACA foil is capable of producing large and stable amounts of lift in the laminar regime.

- **`9_Conduct`**. This example shows how LLALBM can be set up to simulate flow in a closed conduct, again utilising the lattice reading functionalities to introduce a conduct made of obstacle nodes. Nonetheless, although being more cumbersome, it would be possible to directly generate such domain using methods exposed by **`ConstructionInfo`**. The simulation result is shown by Figure 19.

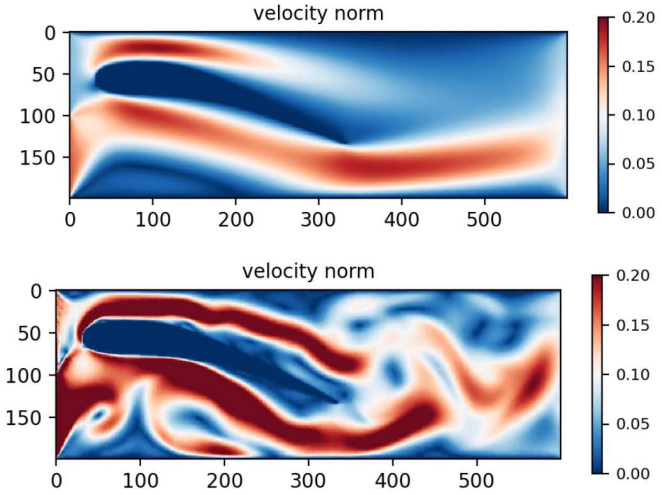

Fig. 17. Flow around a NACA foil during takeoff in laminar(above) and turbulent(below) regimes.

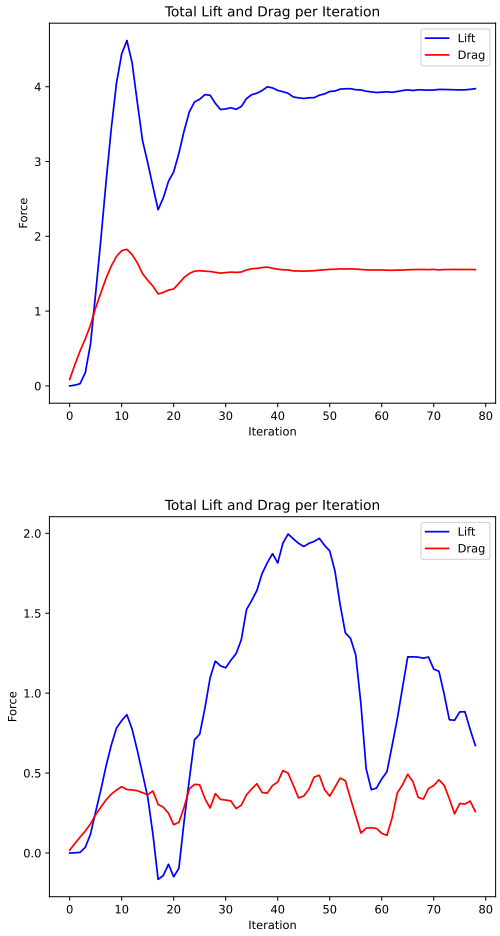

Fig. 18. Lift and Drag forces on a NACA foil during takeoff in laminar (above) and turbulent (below) flow.

Parallelization results.

Parallelization results are presented with respect to the Lid driven cavity simulation. As expected, parallelizing the application is beneficial. A 100×100 Lid-driven-cavity simulation, performed for 3000 time steps without printing is executed in $189.408s$ serially and in $18.757s$ using all the 14 hardware threads of a INTEL I9-13900H processor, yielding a $10.09\times$ speedup. It comes without any doubt that having access to more powerful architectures would yield even higher speedups. Nonetheless, the OpenMP parallelization presents thread management overhead as shown by Figure 20, as the implementation is not perfectly weakly scalable. Using

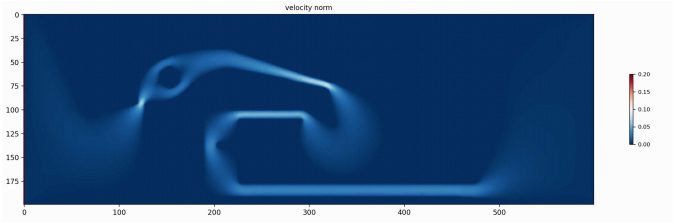

Fig. 19. Flow inside a conduct, with a mesh read from an input file.

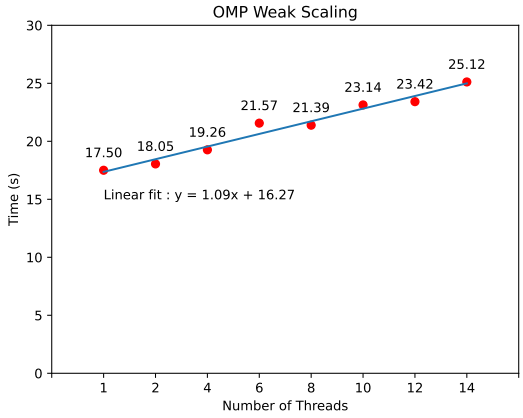

Fig. 20. Weak scalability analysis for the OMP parallelization.

`std::execution` policies is notably slower that OpenMP, most probably due to the internal management. Using OpenACC produces correct simulations, but its application is not efficient due to the lack of specialized hardware.

Possible Extensions

It's possible to extend LLALBM with various features. Here we provide different examples, but there are many other ways to enlarge the program:

- Adding support to more complex velocity sets, together with the specialization of the template for 3D cases, in order to allow the program to capture more complex physical behaviours;
- Adding more Parallelization Policies, i.e. CUDA and MPI, to reach better performances;
- Adding different algorithm techniques to calculate populations, to handle boundaries etc, allowing the program to be more accurate when performing specific tasks.

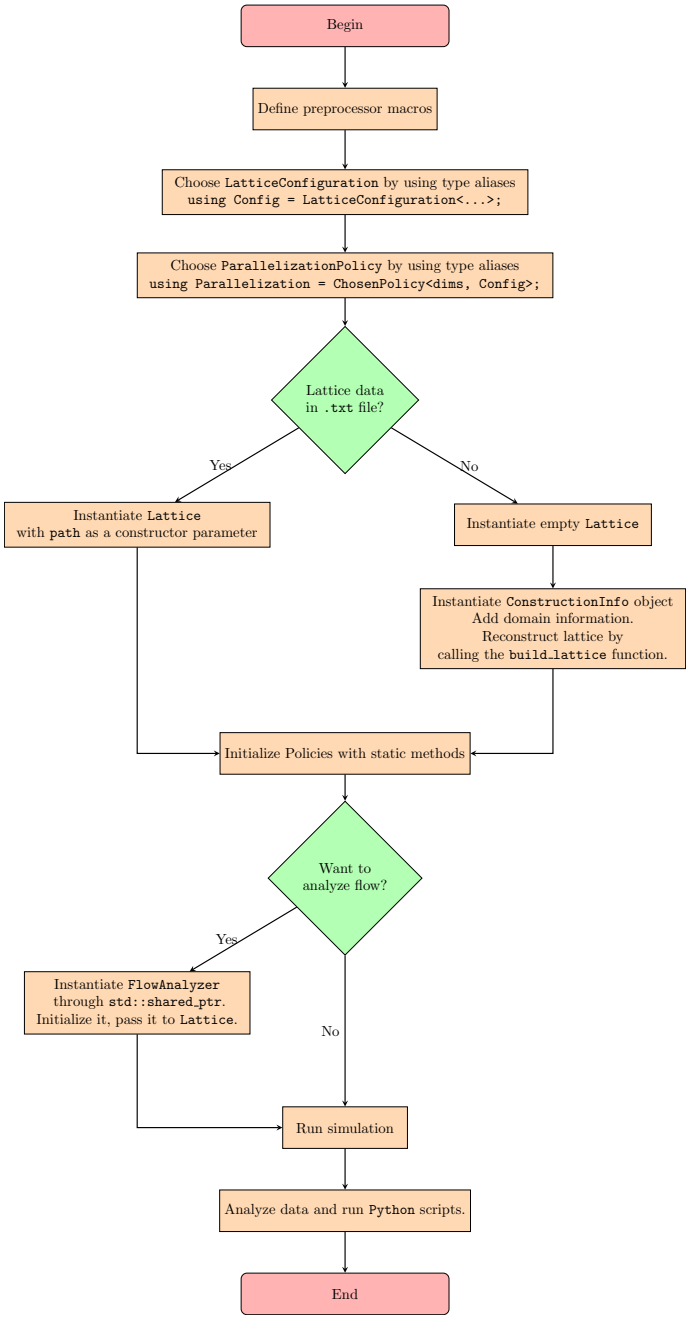

Fig. 21. Execution flow for LBM simulation.

REFERENCES

[1] U Ghia, K.N Ghia, and C.T Shin. "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method". In: *Journal of Computational Physics* 48.3 (1982), pp. 387–411. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(82)90058-4. URL: https://www.sciencedirect.com/science/article/pii/0021999182900584.

[2] Francesco Marson et al. "Enhanced single-node lattice Boltzmann boundary condition for fluid flows". In: *Physical Review E* 103.5 (May 2021). ISSN: 2470-0053. DOI: 10.1103/physreve.103.053308. URL: http://dx.doi.org/10.1103/PhysRevE.103.053308.

[3] Bhatnagar P.L., Gross E.P., and Krook M. "A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems." In: *Physical Review* 94 (1954), pp. 511–525. DOI: http://dx.doi.org/10.1103/PhysRev.94.511.

[4] Zhifeng Yan et al. "Two-relaxation-time lattice Boltzmann method and its application to advective-diffusive-reactive transport". In: *Advances in Water Resources* 109 (2017), pp. 333–342. ISSN: 0309-1708. DOI: https://doi.org/10.1016/j.advwatres.2017.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0309170816307680.

[5] Qisu Zou and Xiaoyi He. "On pressure and velocity boundary conditions for the lattice Boltzmann BGK model". In: *Physics of Fluids* 9.6 (June 1997), pp. 1591–1598. ISSN: 1089-7666. DOI: 10.1063/1.869307. URL: http://dx.doi.org/10.1063/1.869307.