Using openLB for Solving Advection-Diffusion-Reaction Flows in Complex Media

Brage Brevig

July 24, 2024

1 Important Structures in openLB

1.1 Application Outline

Before exploring the features and possibilities of the pre-defined openLB library further, it is necessary to outline the typical application structure when using openLB.

```
int main( int argc, char* argv[] )
{
  // === 1st Step: Initialization ===
  olbInit( &argc, &argv );

  // === 2rd Step: Prepare Geometry ===
  prepareGeometry();

  // === 3rd Step: Prepare Lattice === //

  prepareLattice();

  // === 4th Step: Main Loop with Timer === //

  for ( std::size_t iT = 0; iT < maxSimulationTime; ++iT ) {
    // === 5th Step: Definition of Initial and Boundary Conditions ===
    setBoundaryValues();

    // === 6th Step: Collide and Stream Execution ===
    collideAndStream();

    // === 7th Step: Computation and Output of the Results ===
    getResults();
  }
}
```

Listing 1: Main Body Structure of openLB Applications

Please note that code in Listing 1 is purely for showcase and is not representative of a functioning openLB application. It is completely void of calls to superclasses and general definitions. It however represents the most important features of the openLB software. Applications should, in following order, include on a minimum

1. A definition of the computational domain through a user-defined function such as `prepareGeometry()`
2. A definition of the computational lattice through a user-defined function such as `prepareLattice()`
3. A call to user-defined initial - and boundary conditions as in `setBoundaryValues()`
4. A call to perform the actual computation which in our case is the collision and streaming process as defined by the Lattice Boltzmann Method through `collideAndStream()`
5. A call to a function governing the outputs of the application, such as `getResults()`

Note already here that openLB operates on a premise of making singular calls to the collision - and streaming execution. This means in theory that the user could make changes within the main timer loop in order to change desired features in the temporal dimension. This is very important in light of making changes to the fields and/or the geometries and boundaries of the simulation domain.

1.2 Geometries and Lattices

One of openLB's many out-of-the-ordinary features is the `superGeometry` class. This interface provides tools for an array of crucial simulation operations, such as defining the computational geometry, assigning computational nodes (fluid/solid/boundary etc.) and applying boundaries. However, it is noteworthy that the level of sophistication of this geometry class creates a 'barrier to entry' for those who are new to openLB, as it can be difficult to work with at first. I highly recommend any user of openLB to refer to their official user-guide to better understand the creation of complex geometries such as flow networks, large porous materials or pipe-and-nozzle systems.

Here I present two common ways to define and create geometries in openLB - by manually constructing each part of a geometry using predefined functions within the `superGeometry` class or by using an STL-file reader.

```
1   /// === Step: Prepare Geometry ===
2   std::vector <T> extend(3, T()); // lengths
3   std::vector <T> origin(3, T()); // coordiante origin
4
5   extend[0] = converter.getCharPhysLength(); // x-dir
6   extend[1] = converter.getCharPhysLength(); // y-dir
7   extend[2] = converter.getCharPhysLength(); // z-dir
8
9   origin[0] = -converter.getCharPhysLength() / 2;
10  origin[1] = -converter.getCharPhysLength() / 2;
11  origin[2] = -converter.getCharPhysLength() / 2;
12
13  // function cuboid() creates cuboid with specified
14  // lenghts and origin
15  IndicatorCuboid3D <T> cuboid(extend, origin);
```

Listing 2: User-defined 3D cube geometry in openLB

In Listing 2 we notice line 15 which holds the geometry object `cuboid()` with given dimensions `extend`. This object is constructed using the `IndicatorCuboid3D` class, and generally `IndicatorF(2/3)D` classes exist for spheres, cylinders and more in both 2D and 3D. More intricate designs can be defined by using such indicator classes and basic arithmetic operators (+ and -).

```
1   /// Instantiation of an empty cuboidGeometry
2   #ifdef PARALLEL_MODE_MPI
3       const int noOfCuboids = singleton::mpi().getSize();
4   #else
5       const int noOfCuboids = 2;
6   #endif
7   CuboidGeometry3D <T> cuboidGeometry(cuboid,converter.getPhysDeltaX(),noOfCuboids);
8   cuboidGeometry.setPeriodicity(true, true, true);
9
10  /// Instantiation of a loadBalancer
11  HeuristicLoadBalancer <T> loadBalancer(cuboidGeometry);
12
13  /// Instantiation of a superGeometry
14  SuperGeometry<T,3> superGeometry(cuboidGeometry, loadBalancer, 2);
15  prepareGeometry(superGeometry, cuboid);
```

Listing 3: Creating the parallel computational domains

The snippet in Listing 2 shows only an abstract construct which holds the information of a cuboid with a given physical extension - this geometry defines the entire computational domain. Moving forward, we are interested in showing how to define the subdomains on which the later computational lattice will be distributed. Should the simulation geometry be that of a F1 race car, this snippet shows the method for partitioning that geometry into a given number of computational cuboids (not to be confused with the original cuboid geometry as defined in Listing 2). Each cuboid can be worked

on in parallel. The `cuboidGeometry` can then be sent to a `loadBalancer()` to fairly distribute the work load of the simulation between processors. As an example, the `HeuristicLoadBalancer` class is in alignment with a volumetric LBM approach. Finally, the `superGeometry` class instantiates the computational geometry, with cuboids for parallel computation and the preferred load balancing.

In Listing 3, the `superGeometry` is passed to a user-defined function `prepareGeometry()`

```
    // geometry preparation for cube
void prepareGeometry(SuperGeometry <T,3> &superGeometry,
                     IndicatorF3D <T> &indicator) {
    OstreamManager clout(std::cout, "prepareGeometry");
    clout << "Prepare Geometry ..." << std::endl;

    superGeometry.rename(0, 1);  // labelling of computational nodes
    superGeometry.communicate(); // communication across cuboid boundaries
    superGeometry.clean();       // sweep to clean the geometry of unecessary nodes
    superGeometry.print();

    clout << "Prepare Geometry ... OK" << std::endl;
}
```

Listing 4: More functions related to the superGeometry class

As can be seen from Listing 4, such user-defined functions often hold more functions related to the `superGeometry` class such as the `rename()` function. This function is crucial for determining the distribution of so called material numbers in the computation - in the context of openLB, material numbers are indicators pointing toward the 'type' of computational node. A fluid node, a solid node and a boundary node could all have different material numbers. Setting these material numbers correctly are paramount for the interaction (or no interaction) between the nodes. In Listing 4, the `rename()` function is used to set fluid nodes on the entirety of the interior of the solid geometry.

```
    // Set material number for inflow
  extend[0] = 2.*L;
  origin[0] = -L;
  IndicatorCuboid2D<T> inflow( extend, origin );
  superGeometry.rename( 2,3,1,inflow );
  // Set material number for outflow
  origin[0] = lengthX-L;
  IndicatorCuboid2D<T> outflow( extend, origin );
  superGeometry.rename( 2,4,1,outflow );
  // Set material number for cylinder
  superGeometry.rename( 1,5, circle );
```

Listing 5: More advanced determination of material numbers

Material numbers provide a very good interface for selecting and defining different sections of a solid geometry. For simulations in complex media, a good understanding of these material numbers are very important.

Next, we must define the computational lattice, the mesh of nodes. We begin simply in the main function by setting

```
    // === 3rd Step: Prepare Lattice ===
  SuperLattice<T, NSDESCRIPTOR> NSLattice(superGeometry);
```

in which the `NSDESCRIPTOR` is chosen in the start of the application - more on this in the next section. This superclass is fed to a user-defined function `prepareLatticeNS()` like we did for the `superGeometry`.

```
    void prepareLatticeNS(SuperLattice<T, NSDESCRIPTOR> &NSLattice,
                          UnitConverter<T, NSDESCRIPTOR> const &converter,
                          SuperGeometry<T, 3> &superGeometry,
                          T omega) {
  OstreamManager clout(std::cout, "prepareLatticeNS");
  clout << "Prepare NSE Lattice ..." << std::endl;

  // Material=1 --> bulk dynamics
  // For material 1 (the fluid), set bulk dynamics which is
  // a pre-defined dynamics setting
  NSLattice.defineDynamics<bulkDynamics>(superGeometry, 1);
```

```
12
13    // Initial conditions
14    AnalyticalConst3D<T, T> rhoFluid(1.); // set fluid density
15    AnalyticalConst3D<T, T> u0(0.0, 0.0, 0.0); // set fluid velocity profile
16
17    // set bulkIndicator as the fluid indicator
18    auto bulkIndicator = superGeometry.getMaterialIndicator({0, 1});
19
20    // Initialize all values of distribution functions to their local equilibrium
21    NSLattice.defineRhoU(bulkIndicator, rhoFluid, u0);
22    NSLattice.iniEquilibrium(bulkIndicator, rhoFluid, u0);
23
24    // Lattice initialize with relaxation frequency OMEGA
25    NSLattice.setParameter<descriptors::OMEGA>(omega);
26    NSLattice.initialize();
27
28    {
29      // Examples of possible communication with other fields
30      auto &communicator = NSLattice.getCommunicator(stage::Full());
31      communicator.requestField<descriptors::VELOCITY>();
32      communicator.requestOverlap(NSLattice.getOverlap());
33      communicator.exchangeRequests();
34    }
35
36    clout << "Prepare NSE Lattice ... OK" << std::endl;
37  }
```

Listing 6: Defining and creating the computational lattice for a Navier-Stokes field

Listing 6 provides an example of how to initialize the computational lattice with initial values and a chosen dynamics setting.

1.3 Fields and Dynamics

As mentioned above, the `NSDESCRIPTOR` is a field indicator chosen by the user at the begining of an openLB application. For each field in a program, a descriptor must be chosen. These are classes at the very core of openLB, describing the complexity of the LBM field - these are the D2Q9 and D3Q27 fields which are possible to modify and customize to a great extent. As an example, we define two fields

```
1     typedef D3Q19<VELOCITY> NSDESCRIPTOR;
2     typedef D3Q7<VELOCITY, SOURCE> ADEDESCRIPTOR;
```

Listing 7: Setting descriptors in openLB

The first line sets a descriptor for a D3Q19 field which is meant to describe the Navier-Stokes equations (fluid field) with velocity as its indicator. The velocity indicator is here passed as a template to the D3Q19 template generator. The second field, the advection-diffusion-equation field, is sat with a lower-resolution D3Q7 descriptor and indicated by velocity and a source term which both are passed as parameters to the D3Q7 template.

Much like one can choose the field descriptor, one easily chooses the dynamics that should govern a field as presented in Listing 6. openLB hosts numerous settings for field dynamics, including several BKG-dynamics, Shan-Chen dynamics for multiphase flows and Smagorinsky models. As an example, one can define

```
1     using bulkDynamics = BGKdynamics<T, NSDESCRIPTOR>;
2     using bulkDynamicsAD = SourcedAdvectionDiffusionBGKdynamics<T, ADEDESCRIPTOR>;
```

Listing 8: Setting dynamics for different fields

2 Provided Example: reactingCylinder

In an attempt to start building an application which solves reactive fluid flows in complex media, I have built a program taking inspiration from many other examples available in the `openlB/examples` folder. In this program, a fluid flows around a cylindrical object in a square pipe channel. The fluid reacts at

the surface of the cylinder, and the concentration field is governed by advection-diffusion dynamics. Visualization provides evidence that the simulation works as expected. Changing the Peclét number in the simulation dramatically changes the diffusive and advective behaviour of the concentration field. However, this only serves as a starting place for a simulation which couples the concentration field with a fluid field, such that the concentration field may also be transported by the fluid.

3 Considerations for Multiphysics Simulation Using openLB

As this report deals with the possibility of performing advection-diffusion-reaction flows in complex media, I see it as necessary to make some remarks on how multiphysics can be dealt with in openLB. So far, what we have seen is that

1. An arbitrary number of fields can be created and specialized through calls such as
   `typedef D3Q7<VELOCITY, SOURCE> ADEDESCRIPTOR`
2. These fields can be applied to, and computed in time on lattices which are wrapped on geometries of arbitrary complexity
3. openLB hosts a great deal of potentials and interaction templates which can be useful for physical artifacts such as adsorption, particle-particle interactions, two-phase flows and more.

The main obstacles for the type of multiphysical simulation we wish to arrive at in this project is still at the time of writing contained to

1. Correctly and efficiently implement a coupling operator between fields in a simulation which allows for feedback between said fields, e.g. between a fluid field and a concentration field.
2. Exploring and understanding ways to change the boundaries of a simulated geometry in time, e.g. adding or deleting solid nodes on the computational lattice during crystallisation/precipitation processes.
3. Exploring each setting for fields and dynamics in detail to ensure seamless implementation of multi-field simulations.