User's Guide for `ficsion`

L. Mountrakis

Contents

What is ficsion

ficsion stands for framework for immersed cell suspension. It is a fully parallel suspension solver, based on the open source C++ lattice Boltzmann solver palabos, mainly built for simulating blood suspensions.

ficsion has been developed as a *framework* and not as a single monolithic application. The user/developer can choose the geometry and type of flow he/she will use, the type of cells (red blood cells, platelets, etc.), the model that will be applied to them and a variety of actions: from applying forces to a certain number of surface particles to letting platelets stick to the walls or between each other. In that sense, the already implemented files serve as examples where the user can take the modules and built his/her own application, like lego-bricks.

In case of using ficsion, please cite:

- *Mountrakis, L., Lorenz, E., Malaspinas, O., Alowayyed, S., Chopard, B., Hoekstra, A. G.*, Jul. 2015. **Parallel performance of an IB-LBM suspension simulation framework.** Journal of Computational Science 9, 45-50. URL

Building `ficsion`

Prerequisites

`ficsion` is built on top of `palabos` and it should be possible to be build on any GNU/Linux system with at least the GNU GCC compiler suite and OpenMPI. Make and SCONS are used as the build automation tools. GCC versions 4.7.2 and ... as well as OpenMPI version 1.4.5 and 1.6.5 have been successfully tested. It is very likely that other compilers will also produce favorable results. `ficsion` has been successfully built and ran on x64 and IBM BG/Q systems.

`palabos`

The working version of `palabos` is v1.4r1. Later versions are known to break the initialization of particles and until this issue has been resolved, it is suggested to stick with v1.4r1.

HDF5

`ficsion` uses the `hdf5` library for the output and post-processing of the results. Debian packeges `h5utils hdf5-tools libhdf5-serial-dev` are known to work for the I/O.

Post-processing

Almost all the post-processing is performed by python scripts. The necessary libraries are `numpy`, `matplotlib` and `h5py`.

Debian packages

```
sudo apt-get install gcc g++ gdb make scons
sudo apt-get install openmpi-bin openmpi-checkpoint openmpi-common libopenmpi-dev
sudo apt-get install h5utils hdf5-tools libhdf5-serial-dev
sudo apt-get install python python-numpy python-matplotlib python-h5py
```

Instructions

Ensure that the above packages are downloaded and installed on your system. Assuming that ficsion lies in the directory `${FICSION}`, e.g.

```
export FICSION=${HOME}/ficsion
```

download `palabos` v1.4 from its website and extract it to the path `${FICSION}/palabos`. Now to build `ficsion` just run:

```
make
```

and it will produce an executable with the name `ficsional`. Should you want to change debugging or profiling options, please alter the `Makefile`. Follow a similar approach to compile the examples.

Known issues

- `palabos` version 1.5rX has an issue with particle initialization and produces segmentation faults.
- `ficsion` compiles just fine on MacOSX v10.11 (El Capitain), yet it break when a case is initiated.
- Ordinary `Makefile`s do not work for IBM BG/Q systems. Instead use the `Makefile.cineca` as a template, found in the parent directory.

Getting started with `ficsion`

Before trying to understand `ficsion`, you should try to understand `palabos`. Starting from the `palabos` user guide is a good idea. In short `palabos`, as a parallel 3D CFD solver, uses domain-decomposition to distribute its data among the available processes, following a SIMD (single instruction, multiple data) approach. Parallelism is hidden, in order to ensure that intrusive programming will not harm or hinder obtaining the favorable results of the program, aside from increasing its complexity. It is possible to extend `palabos` without modifying its core, as has been done with `ficsion`. This was performed by developing and overloading essential `palabos` functionalities in the form of data processors, while the source code of `palabos` has not been change.

`ficsion` is built as a framework and comes with various example cases, which the user can alter or find the appropriate modules to built his/her own application. The file `ficsional.cpp` is considered as the "main" file, yet it constitutes one large file trying to incorporate everything, from channel and shear flow to blood suspension and single RBC stretching, which has been proven highly dysfunctional. Yet, this file is kept because of its plurality in the modules it uses, serving as a reference point.

In this section we will describe the constituents and the behavior of a generic `ficsion` case.

Compiling

`ficsion` uses the standard building tools of `palabos`: `make` and `scons`. inside each `Makefile` there are some important parameters. Some of them are:

- `ficsionRoot`: The path of `ficsion` where `ficsion.h` is located.
- `palabosRoot`: Usually set to `$(ficsionRoot)/palabos`, and is advised to be kept this way.
- `projectFiles`: Name of the application you want to compile.
- `debug`: Initially set to `false`, can be set to `true` to debug and test an application. Usually produces much slower code.
- `profile`: Same as the above but for profiling.

`Makefile` has been parameterized to distinguish between MacOS X and Linux and pass the appropriate parameters. Also, the `hdf5` parameters are being passed accordingly, but their order has to be changed when running at an IBM BG/Q machine (along with adding a `-DPLB_BGP` to `compileFlags` as suggested by `palabos`). See `Makefile.cineca` for more information.

The XML file (simulation parameters)

All the parameters of the XML file, excluding switches and the ones defined in `sim`, are either in SI unit, or in dimensionless (non-LB) units. A short explanation of each parameter lies near the parameter in the file. The parameters are read and passed to the main program via the function `readFicsionXML()`. Before they are used, new variables are defined in lattice units (LU).

The field `cellModel` contains all the parameters relevant for the definition of a single RBC. A Shape-Memory RBC is the norm in ficsion (`rbcModel=0`), where all the equilibrium quantities of the model are taken from the initial shape. The mesh of that initial shape is created by mapping the vertices of a sphere created by the subsequent subdivision of either an octahedron or a dodecahedron (relevant parameters: `shape`, `radius`, `minNumOfTriangles`) to that of a surface of an RBC, as obtained by an analytical function.

Parameter `flowType` is a switch that defines the type of flow or of the experiment (e.g. channel flow, Couette, RBC stretching etc) and chooses the appropriate parameter for the definition of the experiment (for our example `Re`, `shearRate` or `stretchForce` respectively). The output frequency `tmeas` and the time of the simulation `tmax` are the only parameters defined in LU. With the standard set of parameters, $\Delta x = 1\mu m$ and $\Delta t \approx 0.1\mu s$.

Many of the parameters in the XML are obsolete in the `examples` and are remainders of `ficsion`s growth. XML files grew alongside `ficsional.cpp` as a combination to catch all possible cases. As `ficsional.cpp` grew, so did its complexity and it became apparent that it has to be broken to several parts, the `examples`. However, the structure of the XML did not change, making XML files compatible with each other (apart from the ones found in `examples/thrombosis`), a not that much wanted side-effect. Often, function `readFicsionXML()` disregards or changes many of the variables defined in the XML file, like `tmax`, `tmeas`, `Re`, `shearRate` or `lx`, `ly`,`lz`.

Running

An application that uses `ficsion` takes as an argument the XML file with the parameters of the simulation. For example, `ficsional` can be run with the following two ways:

- `$ ./ficsional config.xml`
- `$ mpiexec -n ${NUM_PROCS} ./ficsional config.xml`

The first one runs in a single CPU and the latter in `${NUM_PROCS}`. The application immediately creates two folders in the current directory: `log` and `tmp`. Directory `log` contains information on the simulation, as well as running times and performance. Directory `tmp` contains the checkpoint files for the fluid and particles along with the description files, and subdirectory `hdf5` contains the output of the simulation.

`CellField3D` initialization

The initial conditions modules are not perfect and may break the simulation. The options are:

- `RBCField.initialize(cellsOrigin)`, where `cellsOrigin` are the centers of the cells (its type is `vector<Array<T,3> >`) and cells are placed in these positions.
- `randomPositionCellFieldsForGrowth3D(cellFields)` for initializing only the first field (usually RBCs) at semi-random positions and randomly rotated. Cells are deflated to pack higher volume fractions. It results in high initialization time until they are fully inflated, which is many times hindered by the coarse resolution of the mesh in combination with the IBM.
- `orderedPositionCellField3D(cellFields, latticeSize)` initializes one cell field (e.g. RBCs) in a very ordered fashion.
- `orderedPositionMultipleCellField3D(cellFields)` for initializing multiple cells fields in an ordered fashion (very symmetrical IC).

For multiples cells, the latter is used.

I/O

ficsion's output is in the portable format of hdf5. The h5 output files are produced one per output per processor with the naming convention ${TypeIdentifier}.${LB_timestep}.p.${processorId}.h5. The following files may be created:

- Fluid.00000000.p.000.h5 containing information on the velocity, density and force of the LBM field.
- RBC.00000000.p.000.h5 containing all the information of all the SurfaceParticle3D in the particular field. The name is acquired from the identifier given on the creation of CellField3D<T, DESCRIPTOR>.
- RBC_Cell3D.00000000.p.000.h5 containing collective information on, e.g., the RBC as a RBC, like cell velocity, position of the center etc.
- BoundaryParticles.00000000.p.000.h5 contains the positions of the particles that are placed inside the bounce-back boundaries. It is not always written. (See examples/thrombosis/obstacle/obstacle.cpp for more information)
- BondFieldParticles.00000000.p.000.h5 is only output if the case is related with the trombosit module and creates the position and duration of a bond. (See obstacle.cpp for more information)

The user can easily skim the contents of these files with the command h5dump or h5dump -H for only getting the headers on a specific hdf5 file.

Checkpointing

ficsion uses the functionality of palabos for checkpointing both particles and fluid, in addition to creating a checkpoint.xml file with all the characteristics of the simulation. A series of *{.old} files are created to avoid loss of data in case the simulation breaks down during the checkpointing (e.g. due to insufficient disk space). For each field, fluid or particles, a .plb and a .dat are created. The name for the particles is acquired from the identifier given on the creation of CellField3D<T, DESCRIPTOR>. To restart the simulation from a given checkpoint, just use the checkpoint.xml as an argument and make sure that the .plb and .dat files are in the tmp directory. For example:

```
$ ./ficsional tmp/checkpoint.xml
```

Of course by changing the parameters on checkpoint.xml, the simulation can be restarted with these new parameters.

Post-processing and visualizing

HDF5 files are not directly read by Paraview and wrapper files with the XDMF format have to be created in order to visualize the results. A series of scripts have been created to realize this. Up until v4.4, Paraview was inverting the X and Z axis of XDMF files for the fluid and since visualization was more important, the fluid h5 files were also inverting X and Z directions. This can be seen in the snippet below:

```
bool invertXZ_for_XDMF=true;
writeHDF5(lattice, parameters, initIter, invertXZ_for_XDMF);
```

The default value for invertXZ_for_XDMF is false, but up until recently it was set to true. However, every fluid h5 file contain an attribute XZ_inverted_for_XDMF which describes the order of the axes.

Scripts

- `CellHDF5toXMF.py`: Creates XDMF files for cells (RBCs, PLTs etc). Takes on multiple identifiers as arguments. The path has to be `./hdf5/../` (the parent of `./hdf5/`).
- `HDF5toXMF.py`: Creates XDMF files for fluid. The path has to be `./hdf5/../` (the parent of `./hdf5/`).
- `ParticleField3D_HDF5toXMF.py`: Creates XDMF files for individual particles, like `BondParticles`. The path has to be `./hdf5/../` (the parent of `./hdf5/`).
- `batchPostProcess.sh`: Batch script that combines all the above scripts for all known cases (not the most efficient). If an argument is provided, the existing `.xmf` files will be erased and re-created. The path has to be `./tmp/../` (the parent of `./tmp/`).

Other important scripts

- `ficsion_h5utils.py`: Creates numpy arrays from a `tmp/hdf5` directory. RBCs are the default cell type.
- `conficsion.py`: A script to create a multitude of new XML files by changing one or more parameters (creates combinations). Usage: `conficsion.py myconfig.xml --stretchForce=10 20 30 50`.
- `h5repack-and-replace`: Compresses `h5` files. Initial files are deleted and the new ones follow the convention `${TypeIdentifier}.${LB_timestep}.pgz.${processorId}.h5` (`.pgz.` instead of `.p.`).

Examples

`examples/fromSTL/bounceBack.cpp`

The STL file[1] `tube.stl` is loaded as a geometry and voxelized as bounce-back boundaries. Both RBCs and Platelets are present in the simulation. File `tube.stl` should be in the same directory as the application when it runs.

`examples/IBM/caseA_dragForce.cpp`

A single quasi-rigid sphere is fixed at the center of a fully periodic domain, while the fluid is forced. The drag force of the sphere is measured. The physical relevance of this application is very limited.

`examples/IBM/caseB_viscosity.cpp`

A single quasi-rigid sphere is placed in a position close to the center of the domain, fully periodic from X and Y and bounded from Z direction. Velocity boundary conditions are applied to the top bounded part of the domain, and bounce back in the bottom so that a shear-rate profile would be formed. The bottom boundaries measure the stress excreted by the fluid (including the sphere), thus defining its viscosity. This way the apparent viscosity can be defined and the hydrodynamic radius of the sphere can be measured[2].

[1] `std::string meshFileName="tube.stl";`

[2] See L. Mountrakis, Transport of blood cells studied with fully resolved models, PhD Thesis, 2015

examples/IBM/caseC_shearCollision.cpp

In a Couette-type domain two quasi-rigid spheres are place at distance, slightly dislocated and as time progresses, they evolve to collide (not head on). This experiment was to assess the interaction between two IBM membranes.

examples/IBM/caseD_lubrication.cpp

In a fluid at rest two quasi-rigid spheres are place at distance and as time progresses, they evolve to collide head on. This experiment was to assess the lubrication interaction between two IBM membranes.

examples/margination/margination.cpp

RBCs and platelets are placed between two parallel plates where a pressure gradient is applied, evolving into a parabolic velocity profile.

examples/sickleCell/sickleCell.cpp

Forces are applied to opposite vertices in an RBC, in order to obtain the shape of a sickled Red blood cell, as suggested by *Lei and Karniadakis, 2013, Proceedings of the National Academy of Sciences 110(28):11326-11330*. All parameters are obtained from the aforementioned publication.

examples/stenosis/stenosis.cpp

A steep stenosis domain is created using bounce-back boundaries. RBCs and platelets are present, and a strong cells-wall force is applied[^L] using the `applyWallCellForce<T, DESCRIPTOR>` function.

examples/stretching/stretchCell.cpp

A case for validating the elastic properties of an RBC. The input force is obtained from the field `stretchForce` of the XML file. Additional helper scripts are included to create and post-process the various cases for the stretch.

examples/thrombosis/obstacle/obstacle.cpp

One of the verification tests for the `thrombosis` module. A circular obstacle is placed in the center of a square domain, two platelets are placed in the entrance while the fluid is forced. Platelets are let to establish bonds, adhering or not to the surface of the obstacle. Inside the boundaries of the obstacle `boundaryParticle`s are placed, and are interacting with the `SurfaceParticle`s of the platelet, via the `bondDynamics` class.

examples/thrombosis/onePlateletNearWall/onePlateletNearWall.cpp

One platelet is let close to wall in a shear flow environment and is expected to perform a tumbling motion, attaching and detaching from the surface of the wall. This case is still under development.

`examples/thrombosis/plateletWithRBCsNearWall/plateletWithRBCsNearWall.cpp`

Same case as the above, only with RBCs. This case is still under development.

`examples/thrombosis/twoPlateletsApart/twoPlateletsApart.cpp`

Two platelets are placed at a distanced and are forced to separate, measuring the strength of the bonds. This case is still under development.

`examples/viscosity/measureViscosity.cpp`

Same setup as with the `examples/IBM/caseB_viscosity.cpp`, only with a suspension of RBCs. This case is still under development.

Known issues

- Often, function `readFicsionXML()` disregards or changes many of the variables defined in the XML file, like `tmax`, `tmeas`, `Re`, `shearRate` or `lx`, `ly`,`lz`.
- The `lx`, `ly`,`lz` in the XML files may inconsistently result in size of $lx/dx + 1$, accounting for the bounce-back boundary.
- The initial conditions modules are not perfect and in addition to that, they may break the simulation. For example `orderedPositionMultipleCellField3D(cellFields)` cannot always handle dense suspensions and produces segmentation fault. Try lowering the hematocrit, or changing the dimensions of the domain.
- After sufficient time, RBCs are not smooth and have edges. The problem has been isolated to the bending force and setting `kBend` to `1000e2` solves the problem. However this leads to a stiffer RBC and further validation is necessary. Validation and verification should be performed anew
- If any attribute of `SurfaceParticle3D` in its `serialize` and `unserialize` methods, checkpointing is broken.
- `ficsion` will break for small subdomains, typically the ones having one dimension less than $10\mu m$. This is because in order to reduce a single RBC, information from second order neighbors would be necessary (including the envelopes).
- The domain-decomposition of `palabos` in tubes/STL file is not perfect and results in many idle processors. Should be a better way of doing this.
- Fields (`ParticleField3D`s and fluid) typically have different envelope sizes (ghost-node). The order these fields are passed to functionals (derivatives of `BoxProcessingFunctional3D` for example) defines the relative coordinates of the domain. In general relative coordinates and aligning lattices which have different envelope sizes in `palabos` is not straightforward and should be always checked.
- `palabos`'s profiler for version v1.4 has a bug (profiler class is missing the "envelope-update" field), which creates a heisenbug in `ficsion`: if the profiler is used it may break, but it also may not break. Fixing the bug in `palabos` is possible, but this change should be done in all the machines we are running.
- All boundary nodes, bounce-back and velocity, are recognized as boundary nodes when creating boundary particles with `createBoundaryParticleField3D(lattice)`.
- In case an external STL file is necessary, it should always be available in the expected path (e.g for `tube.stl` or `sickledCell.stl`).
- Loading an `xmf` file which is not up to date with the `h5` files will cause the immediate and unforeseen collapse of Paraview.

- For more bugs and features, please visit the bug tracker :-)

A few words on the implementation

Several of the following parts are taken from:

- *Mountrakis, L.*, Sep. 2015. **Transport of blood cells studied with fully resolved models.** Ph.D. thesis, University of Amsterdam. URL

The code is heavily coupled with `palabos` and uses its parallel data structures and modules for fluid and particles, for domain-decomposition, for checkpointing etc. In that respect, `ficsion` is also *limited* by the limitations of `palabos`, e.g. in domain-decomposition. It is possible however to overload the functions `palabos` is using to achieve the desired goal, as has been done for example with the `SurfaceParticle3D` class, overloading `Particle3D`.

`ficsion` is based on the combined immersed boundary-lattice Boltzmann method (IB-LBM). The solid phase, represented as a mesh of Lagrangian surface points interacting with a *finite element model* (FEM), is coupled to the fluid (which is solved with LBM) with IBM. Based on these facts, one can distinguish 3 relevant entities: (a) the fluid (Eulerian) (b) the surface particles (Lagrangian) and (c) the mesh containing the connectivity of the surface points. One more entity has to be defined if the connectivity of surface particles (code: `SurfaceParticle3D`) does not suffice to define a cell[3], because of the existence of e.g. cell-wide quantities like the volume or the surface where information from all vertices[4] is needed.

Data-wise we could use 4 elementary data-structures to describe our problem:

1. `MultiBlockLattice3D<T, DESCRIPTOR> lattice;` for the LBM fluid.
2. `MultiParticleField3D<DenseParticleField3D<T,DESCRIPTOR> > surfaceParticleField3D;` containing all the `SurfaceParticle3D`
3. `MultiParticleField3D<DenseParticleField3D<T,DESCRIPTOR> > cellParticle3D;` containing the `CellParticle3D` which have all the cell-wide quantities stored.
4. `TriangularSurfaceMesh<T> elementaryCellMesh;` the `palabos` structure containing the mesh[5]

and several operation to act on these data. Some important examples are the ones that reduce cell-wide quantities to data contained in `CellParticle3D` and the computation of the constitutive model (code: `CellModel3D`).

In addition to the data and operations, one has to account for maintaining parallelism in all the steps of the process.

Data structures

The aforementioned data and operations can be grouped in classes and data-structures in a multitude of ways, producing favorable results. The following UML diagram provides a part of the organization of data structures and operations in `ficsion`.

Borrowing some text from Lampros's thesis:

[3] By cell we will refer to the "unit" of the suspension, like the red blood cell. In hard sphere suspensions a "particle" carries a similar meaning.

[4] Vertices and surface particles will be used interchangeably in this guide.

[5] Assuming that every cell of the suspension is the same, the mesh is also the same for all the cells of the same cell-field (code: `CellField3D`).

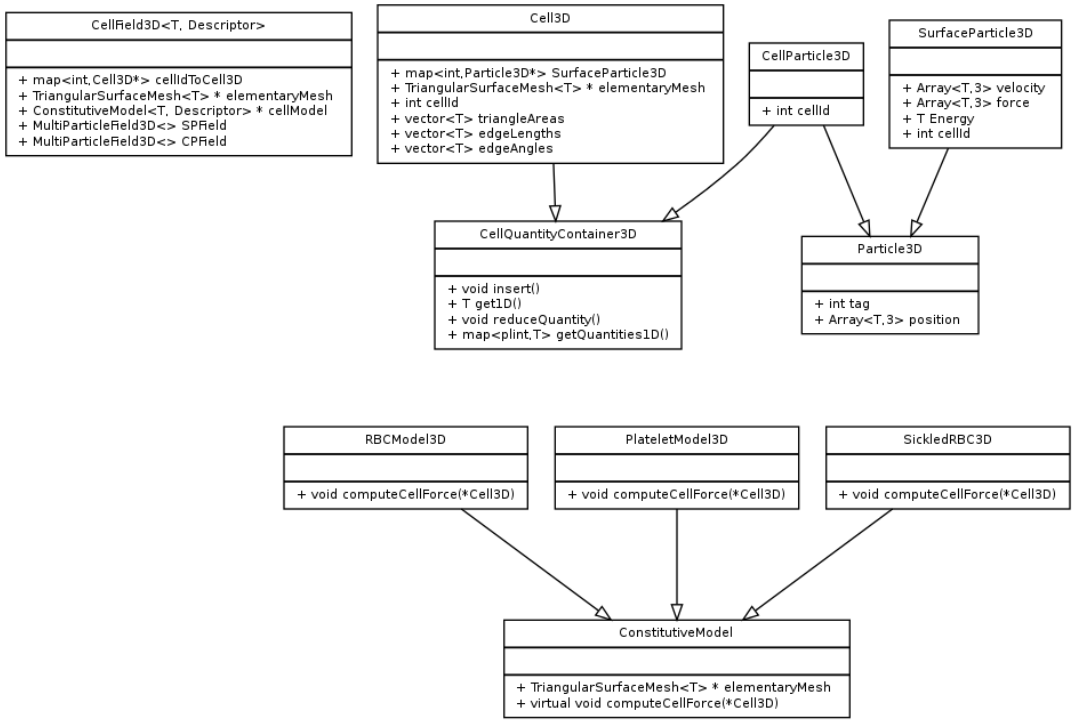

Figure 1: Partial UML diagram. `CellField3D` is a superstructure containing all the necessary data and processes. `Cell3D` does not contain actual surface point data, only references (pointers) to them but contains all the relevant cell-wide operations and data, inheriting from `CellQuantityContainer3D` which is the common parent with `CellParticle3D` taking care of the synchronization between the different domains. `SurfaceParticle3D` inherits from `Particle3D` and adds forces, velocities and other relevant information to it. `ConstitutiveModel` and its children operate on `Cell3D` and contain the constitutive model of the cell. If ones wants to define a different model, one has to overload the `computeCellForce()` method.

The class `MultiParticleField3D` is one of the main `palabos` modules employed in `ficsion`: a container of Lagrangian particles in a structure analogous to the so-called *cell-list* in Molecular Dynamics algorithms. Finding neighboring particles for cell-cell forces can be a time consuming task and cell-lists can reduce the potentially $O(N^2)$ complexity to $O(N)$, which is crucial when the number of particles N becomes large. The parallelism of a `MultiParticleField3D` follows the domain-decomposition of the fluid field and allows to directly identify the particles that belong to the *envelopes* and communicate them to the neighboring subdomains. Envelopes are the boundary nodes communicated in the domain-decomposition approach, also commonly referred to as ghost nodes.

Two instances of `MultiParticleField3D` are utilized for the simulation of a cell-type: one where particles represent the vertices of the cell's surface and interact according to the constitutive model – they are coined `SurfaceParticles3D`– and one for the cell as a whole carrying essential cell information like volume or surface – coined `CellParticles3D`. Since `MultiParticleField3D`s are fully parallelized, the field of `CellParticles3D`. is taking care of all the necessary communication of information between the neighboring subdomains. The position of `CellParticles3D`. is defined as the centroid of the `SurfaceParticle3D` belonging to the actual subdomain (envelopes excluded) and have only a small memory and communication footprint. The drawback of this approach is that it introduces an additional overhead of frequently instantiating and organizing `SurfaceParticles3D` into `CellParticles3D`, due to the motion of cells across the subdomains.

Optimizing communication is an important aspect of the implementation, since the different fields (fluid, `SurfaceParticles3D`, and `CellParticles2D`) have different needs for spatial information over the neighboring subdomains. In that respect, the widths of the envelopes differ for each field, to minimize the amount of data transfered. For the fluid this width is determined by the support width of the IBM kernel ϕ, for the `SurfaceParticles3D` by the maximum distance between two neighboring surface particles of a single cell[6] and for the `CellParticles3D` by the maximum stretch a cell can have. Typically for blood suspensions, `CellParticles3D` have envelopes that are larger than `SurfaceParticles3D`, which in their turn have larger envelopes than the fluid.

With these central structures a cell can freely move between domains and is created upon entrance and destroyed on exit. Additionally, a wide range of cell types can be incorporated within one simulation. The price for this modularity is an overhead created by the bookkeeping and updating of the corresponding data structures.

Workflow

An outline for the parallel workflow for `ficsion` is depicted below. Each task can contribute to a different "group of actions", used for profiling reasons. The workflow is as follows:

Useful modules

- Should the user wants to apply force to the cells, he/she can utilize the module `applyForceToCells(RBCField, RBCellIds, forcesToApply)`, as used in `caseD_lubrication.cpp`.

[6] Owing to the bending force in which two neighboring triangles must be present for the computation, a safe choice would be two time the maximum distance.

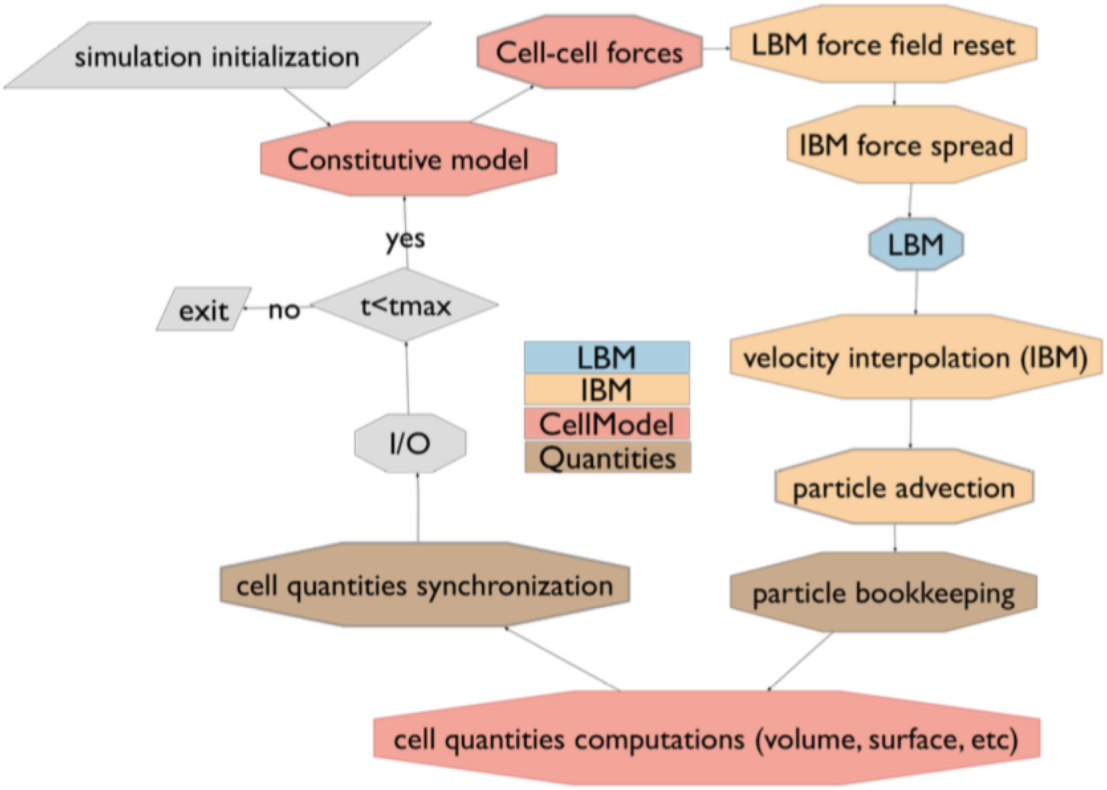

Figure 2: Scheme of the simulation workflow. Boxes with the same color are grouped together and have the same time apportionment in `fcnPerformance.log`. Red is for the constitutive model, blue for LBM, yellow for IBM and brown for the bookkeeping of cells. At the end of each routine a synchronization step is performed.

It is also possible to apply force in only a subset of the vertices of the cells, like `sickleCell.cpp` or `stretchCell.cpp`. The latter two cases use wrapper functions to achieve this, however an overloaded function `applyForceToCells(cellField, cellIds, iVertices, forces)` is available along the aforementioned.
- Class `ProximityDynamics3D` can be overloaded and used with `ApplyProximityDynamics3D` to apply forces (repulsion, attraction, from Morse potential etc) between cells and particles, based on a cutoff distance between them. Examples on how this can be used can be found in `./trombosit/BondField3D.h`.

Performance suggestions

- Synchronization is performed at the end of each functional, increasing the communication cost. Transitioning from `palabos`'s `applyProcessingFunctional` to `integrateProcessingFunctional` will save considerable time.
- `palabos` v1.5 has a better class for Particle fields that are not dense and using the `LightParticleField3D` for `CellParticle3D` (instead of `DenseParticleField3D`) will also improve performance.
- Significant part is spent on book-keeping the particles of `DenseParticleField3D`. Overloading `DenseParticleField3D` with a `map<,>` construct which is updated upon addition and removal of `Particle3D` will also save time.
- Envelope sizes were chosen based on the principle *"validity over performance"* and are not optimized at all.

Thrombosis module

TBA

Contribute

Source

- Source Code: URL
- Issue Tracker: URL

Resources

- `ficsion` performance predictor: URL
- Palabos Documentation: URL
- Palabos User Guide (PDF): URL

License

No known license