A Python extension for the massively parallel framework waLBerla

Martin Bauer*, Florian Schornbaum*, Christian Godenschwager*†, Matthias Markl*, Daniela Anderl*
Harald Köstler* and Ulrich Rüde*

* Friedrich-Alexander-Universität Erlangen-Nürnberg, Cauerstraße 11, Erlangen, Germany
Email: {martin.bauer,florian.schornbaum,matthias.markl,daniela.anderl,harald.koestler,uli.ruede}@fau.de

† Siemens AG, An der Lände 1, 91301 Forchheim
Email: christian.godenschwager.ext@siemens.com

***Abstract*—We present a Python extension to the massively parallel HPC framework WALBERLA. WALBERLA is a framework for stencil based algorithms operating on block-structured grids, with the main application field being fluid simulations in complex geometries using the lattice Boltzmann method. Careful performance engineering results in good scalability to over 400,000 cores. To increase the usability and flexibility of the framework, a Python interface was developed. Python extensions are used at all stages of the simulation pipeline: They simplify and automate scenario setup, evaluation, and plotting. We show how our Python interface outperforms the existing a text-file-based configuration mechanism, providing features like automatic nondimensionalization of physical quantities and handling of complex parameter dependencies. Furthermore, Python is used to process and evaluate results while the simulation is running, leading to smaller output files and the possibility to adjust parameters dependent on the current simulation state. C++ data structures are exported such that a seamless interfacing to other numerical Python libraries is possible. The expressive power of Python and the performance of C++ make development of efficient code with low time effort possible.**

I. Introduction

Many massively parallel codes are written for a specific use-case, making strict assumptions on the scenario being simulated. These restrictions allow the programmer to optimize the code for its specific use-case, exploiting information already available at compile time. This approach is not feasible when developing a general purpose framework targeted at a variety of different applications. While the highest priority is still performance and scalability, at the same time the framework has to be easy to use, modular, and extensible. However, performance and flexibility requirements are not necessarily conflicting goals.

A common approach is to separate compute intensive parts of the program, so-called kernels, and write several versions of them, each one being optimized for a special scenario or a specific target architecture. The framework then selects the kernel which matches the problem and the hardware best. Kernels typically are developed in low-level programming languages like C/C++ or Fortran which allow close control over the hardware. These system programming languages are very powerful but also difficult to learn. The complex and subtle rules often prevent domain experts, who have a limited programming expertise, to use parallel high performance codes. While being the best choice for performance critical portions of the code, these languages are therefore not well suited for other less time critical framework parts, like simulation setup, simulation control, and result evaluation. The run time of these management tasks is usually negligible compared to kernel run times, since they do not have to be executed as often as the compute kernels or are per-se less compute intensive. Therefore, these routines are especially suitable for implementing them in a higher-level language like Python.

We present a Python extension to the massively parallel HPC framework WALBERLA which aims to increase the ease of use and decrease the development time of non time critical functions, implementing tasks like domain setup, simulation control, and result evaluation.

II. WALBERLA FRAMEWORK

WALBERLA is a massively parallel software framework supporting a wide range of applications. Its main application are simulations based on the lattice Boltzmann method (LBM), which is reflected in the acronym "widely applicable lattice Boltzmann solver from Erlangen". Having initially been a LBM framework, WALBERLA evolved over time into a general purpose HPC framework for algorithms that can make use of a block-structured domain decomposition. It offers data structures for implementing stencil based algorithms together with load balancing mechanisms and efficient input and output of simulation data. WALBERLA has two primary design goals: Being efficient and scalable on current supercomputer architectures, while at the same time being flexible and modular enough to support various applications [1], [2].

In this section, we start by giving a short overview of the lattice Boltzmann free surface method, followed by a description of the WALBERLA software stack used to implement the method.

A. Free Surface Lattice Boltzmann Method

The lattice Boltzmann method is a mesoscopic method for solving CFD problems. It is based on a discrete version of

the Boltzmann equation for gases. The continuous Boltzmann equation comes from kinetic theory and reads:

$$\frac{\partial f}{\partial t} + \xi \cdot \nabla f = Q(f, f)$$

with $f(x, \xi, t)$ being the continuous probability density function representing the probability of meeting a particle with velocity ξ at position x at time t. The left hand side of the equation describes the transport of particles, whereas the right hand side $Q(f, f)$ stands for a general particle collision term.

To discretize the velocity space, a D3Q19 stencil, with 19 discrete velocities $\{e_\alpha | \alpha = 0, ..., 18\}$ and corresponding particle distribution functions (PDFs) denoted by $f_\alpha(x, t)$, is used [3] . With a time step length of Δt, the discrete LB evolution equation then reads:

$$f_\alpha(x_i + e_\alpha \Delta t, t + \Delta t) - f_\alpha(x_i, t) = \Omega_\alpha(f)$$

As discrete LBM collision operator $\Omega_\alpha(f)$, a two relaxation time scheme (TRT) is used [4], [5]. The time and space discretization yield an explicit time stepping scheme on a regular grid, which enables efficient parallelization due to the high locality of the scheme. To update the 19 PDFs stored in a cell, only PDFs from the cell itself and neighboring cells are required.

This basic LBM is extended to facilitate the simulation of two-phase flows. The free surface lattice Boltzmann method (FSLBM) is based on the assumption that the simulated liquid-gas flow is completely dominated by the heavier phase such that the dynamics of the lighter gas phase can be neglected. The problem is reduced to a single-phase flow with a free boundary [6]. Following a volume of fluid approach, for each cell a *fill level* φ is stored, representing the volume fraction of the heavier fluid in each cell. The fill level determines the state of a cell: cells entirely filled with heavier fluid ($\varphi = 1$) are marked as a *liquid cell* and are simulated by the LBM described above, whereas in *gas cells* ($\varphi = 0$) no LBM treatment is necessary. Between the two phases, in cells where $0 < \varphi < 1$, a closed layer of so-called *interface cells* is maintained, tracking all cells where the free boundary condition has to be applied. A mass advection algorithm modifies the fill level φ and triggers conversions of cell states.

Additionally, regions of connected gas cells are tracked with a special *bubble model* [7]. It calculates the volume of bubbles, in order to compute the pressure for each gas cell as fraction of current to initial bubble volume. The gas pressure is essential for the treatment of the free boundary. Possible topology changes of bubbles require a sophisticated parallel algorithm to track bubble merges and splits.

B. Software Architecture

WALBERLA is built out of a set of modules, which can be grouped into three layers (Fig. 1). The bottom layer of WALBERLA provides data structures and functions for implementing stencil-based algorithms on block-structured grids and will be described in detail in this section. The second layer consists of specific algorithms which make use of the core layer. WALBERLA is predominantly used for lattice Boltzmann simulations, but also phase field and multigrid methods have been implemented using the framework. The topmost layer is formed by algorithm extensions, like methods for fluid structure interaction [8] or the free surface LBM used in this paper.

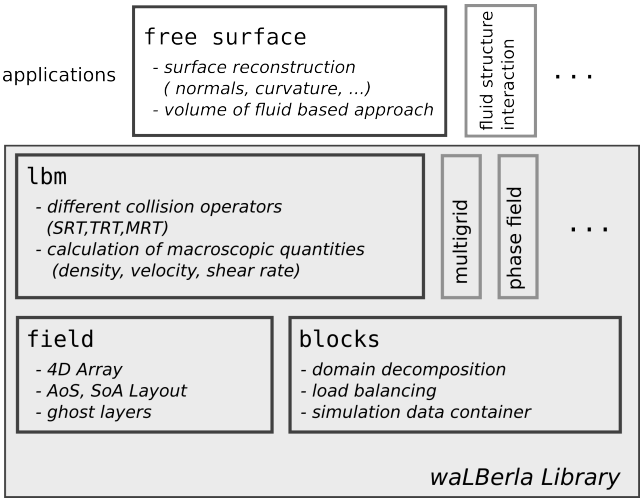

Fig. 1. Overview of Software Architecture

1) Domain Decomposition: For parallel simulations, the domain is partitioned into smaller, equally sized sub-domains called *blocks* which are then distributed to processes. Blocks are not only the basis for parallelization, but also the basic unit of load balancing. Since there might be different computational efforts required to process each block, it is possible to put more than one block on a process, balancing the computation time across all processes.

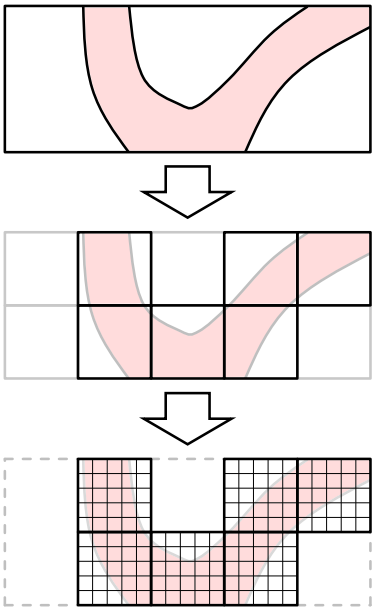

Fig. 2. Domain partitioning: block division and subsequent grid generation.

To illustrate this concept consider a simulation scenario as depicted in Figure 2. In this scenario, the blood flow through an artery tree, which is specified by a triangle surface mesh, has to be simulated. As a first step, the bounding box of the artery tree is decomposed into blocks, then blocks which do not overlap the mesh are discarded. In the second step the blocks

are assigned to processes, taking the computational load and memory requirements of a block into account. In this example, the computational load of a block is proportional to the number of fluid cells contained in it. The load balancing also takes into account neighborhood relations of blocks and the amount of data which has to be communicated between processes.

2) Fields: Besides being the basic unit of load balancing, blocks also act as containers for distributed simulation data structures. In the case of LBM simulations, the main data structure is the lattice. This lattice is fully distributed to all blocks, where the local part of the lattice is represented by an instance of the *field* class (last stage of Fig. 2).

Fields are implemented as four dimensional arrays, three dimensions for space, one dimension to store multiple values per cell. In the LBM case, this forth coordinate is used to store the 19 PDF values. The field abstraction makes it possible to switch between an array-of-structures (AoS) and a structure-of-arrays (SoA) memory layout easily. For many stencil algorithms, a AoS layout is beneficial since in this case all values of a cell are stored consecutively in memory. This data locality results in an efficient usage of caches. However, when optimizing algorithms to make use of SIMD instruction set extensions, usually a SoA layout is better suited. Additionally, operands of SIMD instructions have to be aligned in memory, resulting in the requirement that the first elements of each line are stored at aligned memory locations. To fulfill this restriction, additional space has to be allocated at the end of a coordinate (padding). This is implemented in the field class by discriminating between the requested size of a coordinate and the allocated size of a coordinate. This discrimination is also helpful when implementing sliced views on fields, which operate on the original field data, but have different sizes.

waLBerla offers a synchronization mechanism for fields based on ghost layers. The field is extended by one or more layers to synchronize cell data on the boundary between neighboring blocks. The neighbor access pattern of the stencil algorithm determines the number of required ghost layers: if only next neighbors are accessed as in the LBM case, one ghost layer is sufficient. Accessing cells further away requires more ghost layers.

C. Performance and Scalability

The waLBerla framework has been run on various compute clusters, for example on JUQUEEN in Jülich, on Tsubame at the GSIC Center at the Tokyo Institute of Technology in Japan, and on SuperMUC at LRZ in Munich.

Figure 3 illustrates weak scaling results obtained when running a LBM based fluid simulation on a dense regular domain with waLBerla on SuperMUC in Munich and the JUQUEEN system in Jülich. Further weak and strong scaling results can be found in [9].

III. Python Interface

This section gives an overview of the Python interface to the basic waLBerla data structures described in the previous section.

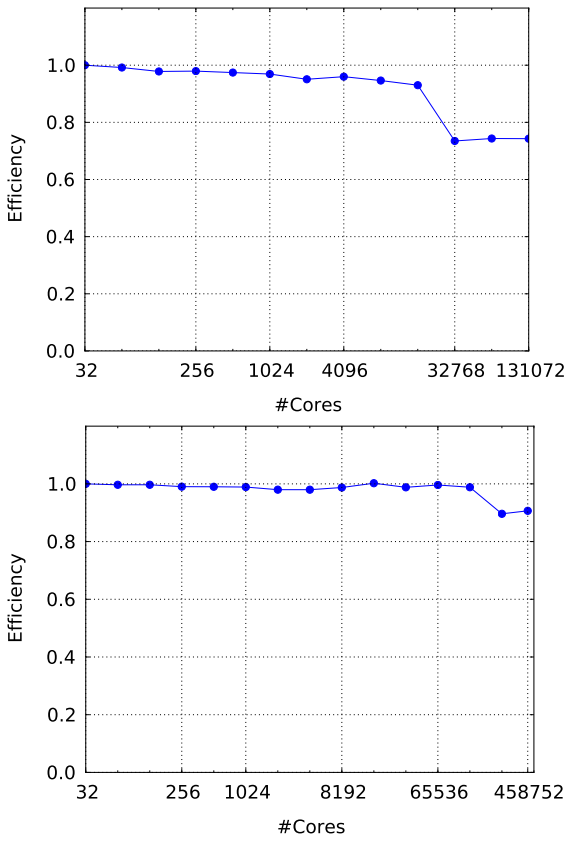

Fig. 3. LBM weak scaling results with waLBerla on SuperMUC (top) and JUQUEEN (bottom)

waLBerla was designed initially as a pure C++ framework, the Python interface is developed as an optional extension to the framework. The motivation for using Python came from the need for a more flexible simulation setup mechanism. A text file was used to configure the simulation, which became increasingly complex over time. Some users wrote Python scripts to create this configuration file, leading to the idea to embed Python directly into waLBerla. It turned out that the embedding of Python is useful, not only for configuration purposes, but also for simulation control and analysis as described in the following section.

The C++ part of waLBerla makes use of various *boost* libraries [10] for example for portable filesystem access, for memory management with smart pointers, and for parsing of input parameters using regular expression. Since waLBerla already depends on *boost*, we also make use of the *boost::python* library [11] to expose our C++ data structures to Python. For certain tasks, however, it was necessary to use the Python C-API directly, since the required functionality is not available in *boost::python*.

There are two mechanisms for coupling C++ and Python. The first approach is to create a Python module as shared library out of the C++ code. Using this solution, the driving code is written in Python, making use of exposed C++ functionality in the library. In the second approach, Python is embedded into the C++ application by linking against *libpython*. When the second approach is used, the simulation is driven by C++

code, optionally calling Python functions at certain stages of the simulation. We choose the second approach, since our main goal is to extend our C++ simulation code making it more flexible and easier to use. However, an implementation of the first approach is currently in development, offering the possibility to drive simulations using Python code.

To interact with C++ simulation code via Python, the user supplies a script file, decorated with callback functions as shown in Listing 1. The code example shows a callback function as it is often used for custom post-processing or monitoring of the current simulation.

```
import waLBerla

@waLBerla.callback( "at_end_of_timestep" )
def my_callback( blockstorage, **kwargs ):
   for block in blockstorage:
      # access and analyse simulation data
      velocity_field = block['velocity']
```

Listing 1. Embedding Python into C++ using callback annotations

In this case, the function is called after a simulation time step has finished, such that all data is in a consistent state. The callback mechanism exposes all simulation data, passing the blockstorage object to the function. For simple and intuitive access, the block collection is exposed as a mapping type, mimicking the behavior of a Python dictionary. In parallel simulations, the Python callback function is invoked on every process, whereas the blockstorage contains only blocks assigned to the current process. Thus, the loop over all blocks is implicitly parallelized. If a global quantity has to be calculated, the data reduction has to be programmed manually using MPI routines.

The C++ counterpart of the callback function is shown in Listing 2. A callback object is created, identified by a string which has to match the decorator string in the Python script. Then the function arguments are passed, either by reference (“exposePtr“) or by value (”exposeCopy“). In order to pass a C++ object to the callback, the class has to be registered for export using mechanisms provided by *boost::python* . For most data-structures, this can be done in a straightforward way, for the field class however, a special approach has to be taken.

```
Callback cb ( "at_end_of_timestep" );
cb.exposePtr("blockstorage", blockStorage );
cb(); // run python function
```

Listing 2. Embedding Python into C++ using callback annotations

A. waLBerla Field as NumPy Array

The field class is one of the central data structures of waLBerla. It is the data structure that end users have to work with the most, for example when setting up simulation geometry and boundary conditions or when evaluating and analyzing the current simulation state. As described above, a field is essentially a four dimensional array supporting different memory layouts (AoS and SoA), aligned allocation strategies and advanced indexing (slices).

A similar data structure widely used in the Python community is `ndarray` of the NumPy package [12]. A wide range of algorithms exist operating on NumPy arrays, for example linear algebra-, Fourier transformation-, or image processing routines. To make use of these algorithms, it is desirable to be able to convert waLBerla fields efficiently into the NumPy representation. Copying data between these representations is not a feasible option due to performance reasons and memory limitations. Simulations are oftentimes set up in a way to fully utilize the available memory of a compute node. Big portions of the allocated memory are occupied by the lattice i.e. the field. Thus, an export mechanism for fields is required, which offers read-write access to the field *without copying data*. The exposed object should behave like a NumPy array such that algorithms from the NumPy and SciPy ecosystem can be used. There are only two options fulfilling these requirements: either to essentially re-implement `ndarray` and to export all functions using mechanisms provided by *boost::python*. This is the *duck-typing* approach popular in Python: the exported field would behave exactly like a `ndarray` and could therefore be used with all algorithms expecting NumPy arrays. Due to the high implementation effort, this approach was not used. Instead, we implement the Python *buffer protocol* [13] which provides a standardized way to directly access memory buffers. This protocol supports advanced memory layouts used by the waLBerla field class through definition of strides and offsets. Among others, NumPy arrays can be constructed from buffer objects, so all requirements can be fulfilled using this approach, without introducing any dependency of waLBerla to the NumPy library. The buffer protocol is not available in *boost::python*, so in this case the C-API of Python had to be used directly.

B. Encountered Difficulties

The two primary goals of waLBerla, being an HPC framework, are flexibility and performance. To achieve both goals, it is necessary to make use of advanced C++ template mechanisms. Exporting these template constructs to Python can be difficult since all templates have to be instantiated with all possible parameter combinations. In situations where this cannot be done manually, the instantiation is done using template meta programming. Instantiating and exporting all possible template parameter combinations would result in long compile times and would increase the size of the executable significantly. Thus a tradeoff has to be made and only the commonly used template parameter combinations are exported to Python. If, however, a user needs other combinations in his application, the framework provides a simple mechanism to configure which template parameter combinations are exported.

IV. Simplification of Simulation Workflow

In this section we demonstrate the usage of the waLBerla Python interface by describing the workflow of setting up a FSLBM scenario. We show how to configure, control, and evaluate the simulation using Python callback functions.

The following example scenario is a two phase flow problem in a rectangular channel. A foam is transported through a channel as depicted in Figure 4. Due to gravity, bubbles rise to the top of the channel, forming a thin liquid film at the bottom. A characteristic flow profile is expected, with a parabolic shape

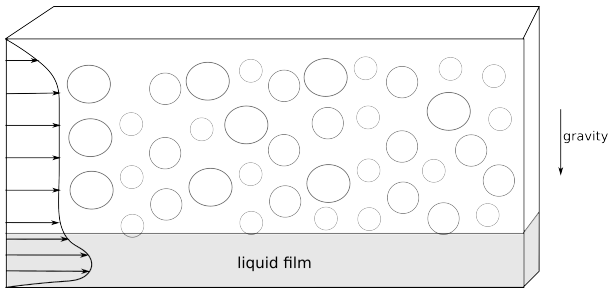

Fig. 4. Flow of foam through channel

in the liquid film, and an almost constant velocity in the rest of the channel. The goal of the simulation is to investigate the stability of the transported foam and its dependence on surface properties and rheological parameters.

Desired output quantities of the simulation are gas fractions and velocities in different parts of the domain. Additionally, the flow profile and foam stability should be evaluated, when the pressure gradient which drives the channel is changed, i.e. when the pressure gradient is switched on and off.

Using this example scenario, we describe how the Python interface simplifies the configuration and geometry setup of a LBM simulation.

A. Simulation Setup

To setup a simulation on a regular grid, there are typically two different kinds of input required. Before the Python interface was developed, the most flexible choice for domain initialization, geometry setup, and specification of boundaries was a voxel-based input file. Such voxel files had to be generated using external tools. Additional configuration like discretization options or physical parameters where given in a second text file. This text file was formatted in a syntax similar to the JavaScript object notation (JSON), providing parameters as a hierarchy of key-value pairs. When setting up LBM simulations, physical parameters have to be converted to nondimensionalized lattice units [14]. The conversion factors depend on the choice of discretization parameters. To simplify this process, the configuration file was extended with special functionality, enabling the user to easily convert physical to lattice units. The nondimensionalization problem is, however, only an instance of a wider range of problems. The problem that parameters are interdependent and that this interdependency should be defined by the user in the configuration file. Parameters can depend on each other in complex ways: consider the time step length, which should be chosen maximal, subject to stability constraints. Complex parameter dependencies can lead to configuration errors. To improve usability, especially for inexperienced users, configuration mistakes should be detected before the simulation runs. Therefore, all parameters have to be checked if they are in a valid range and consistent with other parameters.

Trying to handle these problems in a flexible and user friendly way leads towards more and more custom extensions in the input file, essentially developing a custom scripting language. Instead of pursuing this approach further the decision was made to use an existing scripting language like Python. Python offers libraries that can handle all of the requirements described above. There are libraries available for defining and manipulating physical units, making them suitable for solving nondimensionalization problems [15]. To handle complex parameter dependencies, we use linear algebra and optimization routines from *SciPy* [16]. Also, the ability for symbolic calculations as provided by the *SymPy* library [17], proves useful in a configuration file.

The hierarchical key-value configuration is represented in Python as a nested dictionary object. This dictionary is built up in a specially decorated function called by the C++ part of the framework (first function in Listing 3). For simplicity, the C++ part expects all parameters to be in valid nondimensionalized lattice units. Nondimensionalization and parameter validation is completely done in Python.

The definition of domain geometry and boundary conditions can also be handled using a Python callback function, substituting the previously used voxel file. This callback is executed once for every cell before the simulation starts. Via the returned dictionary object the initial cell state is defined, consisting of initial velocity and density, or of the boundary type and boundary parameters.

```
@waLBerla.callback( "config" )
def config():
   c = {
      'Physical' : {
       'viscosity'       : 1e-6*m*m/s,
       'surface_tension': 0.072*N/m
       'dx'              : 0.01*m,
       # ...
      }
      'Control' : {
        'timesteps' : 10000,
        'vtk_output_interval': 100,
        # ...
      }
   }
   compute_derived_parameters(c)
   c['Physical']['dt'] = find_optimal_dt(c)
   nondimensionalize(c)
   return c


gas_bubbles=dense_sphere_packing(300,100,100)

@waLBerla.callback( "domain_init" )
def geometry_and_boundary_setup( cell ):
   p_w = c['Physics']['pressure_west']
   if   is_at_border( cell, 'W'   ):
      boundary = [ 'pressure', p_w ]
   elif is_at_border( cell, 'E'   ):
      boundary = [ 'pressure', 1.0 ]
   elif is_at_border( cell, 'NSTB'):
      boundary = [ 'noslip' ]
   else:
      boundary = []

   return{'fill_level':1-gas_bubbles.overlap(cell),
          'boundary'  :boundary }
```

Listing 3. Simulation Setup

Listing 3 shows how to set up the channel flow scenario using a Python file. The first callback function substitutes the JSON file, providing parameters as a dictionary. Before passing the parameters to the C++ code, several functions operate on the dictionary, handling nondimensionalization and calculation

of dependent parameters.

The second callback function handles boundary setup and domain initialization. In our example scenario, we prescribe a pressure boundary on the left (east) and right (west) end of the domain, all other borders are set to no-slip boundary conditions. In this example, only boundary conditions at domain borders are set. However, it is possible to set boundaries at arbitrary cells in the domain. The bubbles are placed in the channel as a dense sphere packing, where bubble positions are calculated by a Python function. This routine fills the whole domain with equally sized bubbles. As shown in Listing 3, the initial gas fraction of a cell is set using the initialization callback mechanism.

B. Evaluation

Storing the complete state of a big parallel LBM simulation results in output files with sizes up to several gigabytes. Typically, the complete flow field together with cell fill levels is written to a voxel based file format for analysis. Especially for free surface simulations not all of this detailed output is required. When simulating the behavior of foams, only some higher level information like gas fractions in certain areas or number, shape, and velocity of bubbles are of interest. These quantities can be obtained by a post-processing step using the raw voxel based output of velocity and fill level. For moderately sized simulations, the raw output can be copied to a desktop machine and post-processed using graphical tools like ParaView [18]. For bigger simulations, however, it has many advantages to do the post-processing directly on the cluster where the simulation was run. The time for copying the raw output files can be saved and the post-processing algorithm itself can be parallelized if necessary. Since the requirements of this evaluation step vary widely depending on the scenario at hand, the analysis routines are not included in the core C++ part of the framework. The post-processing is usually done in user written, custom Python scripts which have a lower development time than C++ code. This situation was another motivation to directly couple our C++ simulation framework to Python, such that evaluation scripts can run during the simulation and operate directly on simulation data making the output of raw voxel data obsolete.

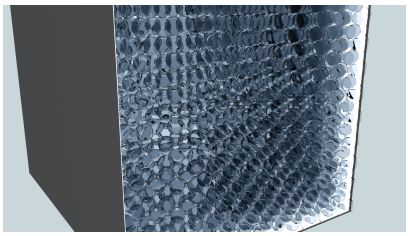

Fig. 5. Cross section of foam flow scenario orthogonal to flow direction

In the evaluation callback functions, we can make use of the exposed C++ data structures. Since the simulation data is fully distributed, also the evaluation has to be done in a distributed way. An example is shown in Listing 4 where the maximum velocity along the flow direction for the channel example problem is calculated. The first iteration goes over all local blocks, extracting the velocity field as NumPy array. As described above, the NumPy array is only a view on the already existing data, no copy is made. Using the velocity field, first the per-process maximum is determined then a global MPI reduce operation has to be done to obtain the global maximum.

```
import numpy as np

@waLBerla.callback( "at_end_of_timestep" )
def evaluation(blockstorage, bubbles):
   # Distributed evaluation
   x_vel_max = 0
   for block in blockstorage:
      vel_field = np.asarray(block['velocity'])
      x_vel_max = max(vel_field[:,:,:,0].max(),
                      x_vel_max)

   x_vel_max = mpi.reduce( x_vel_max, mpi.MAX )
   if x_vel_max: #valid on root only
      log.result("Max X Vel", x_vel_max)


   # Gather and evaluate locally
   size=blockstorage.numberOfCells()
   vel_profile_z=gather_slice(x=size[0]/2,
                              y=size[1]/2,
                              coarsen=4 )
   if vel_profile_z: #valid on root only
      eval_vel_profile(vel_profile_z)
```

Listing 4. Simulation Evaluation

Due to the distributed nature of the data, this evaluation step is still somewhat complex. However, we can simplify it in some cases, especially when working with smaller subsets of the data. Let us for example consider the evaluation of the flow profile in the channel scenario. The velocity profile (as depicted in Figure 4) in the middle of the channel along a line orthogonal to the flow direction is analyzed. From this information we can obtain, for example, the height of the thin liquid layer at the bottom. To simplify the evaluation routine, we first collect this one dimensional dataset on a single process, which is possible since the 1D slice is much smaller than the complete field. In case it is still too big, the slice can also be coarsened, meaning that only every n'th cell is gathered. Then all required data is stored on a single process, enabling a simple serial evaluation of the results.

C. Simulation Control

The Python interface can not only be used for domain setup and evaluation but also to interact with the simulation while it is running using a Python console. This is especially useful during the development process. One can visualize and analyze the simulation state with plotting libraries (e.g. matplotlib [19]) and then modify the simulation state interactively.

In case of a serial program, the Python C-API offers high level functions to start an interactive interpreter loop. For parallel simulations, this approach is not feasible, since every process would start its own console.

Instead, a custom solution was developed, where one designated process runs the interpreter loop, broadcasting the entered commands to all other processes, which are then executed simultaneously. The custom interpreter loop reads the user input line by line, until a full command was entered. This advanced detection has to be done, since Python commands

can span multiple lines. After a full command was detected, it is sent to all other processes using a MPI broadcast operation.

There are two ways to start an interactive console while a simulation is running: in UNIX environments, the user can send a POSIX signal to interrupt the simulation. The simulation continues until the end of the current time step, such that all internal data structures are in a consistent state. Then the Python console is run, using the standard input/output streams. A second method based on TCP sockets can be used when sending POSIX signals is not feasible, for example in Windows environments or when starting a parallel simulation using a job scheduler. In this case, the program listens for TCP connections. When a client connects, the simulation is interrupted after the current time step such that the user can interact with the program using a telnet client.

Besides the ability to modify the simulation state interactively, the modification of parameters can of course also be done automatically. Being able to evaluate the simulation state in Python, we have information available *during* the simulation which previously were acquired in a post-processing step. This information can be used to modify parameters while the simulation is running, effectively implementing a feedback loop.

In the channel flow scenario, this can be used to investigate foam stability when the inflow boundary is switched on or off. Evaluation routines are used to determine if the simulation has reached a steady state, then the driving pressure boundary can be modified.

D. Summary

To summarize, the Python interface has greatly simplified the entire simulation toolchain.

A schematic of a typical workflow of using WALBERLA without the Python extension is depicted in Figure 6. The configuration is supplied in two different files, one voxel file for specifying domain geometry and boundary information and secondly a text based parameter file defining options as a hierarchical set of key-value pairs.

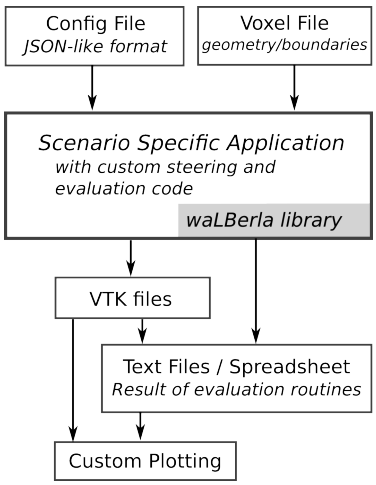

Fig. 6. Typical simulation workflow without usage of Python Interface

The C++ simulation code itself is also tailored to the scenario, including custom steering and evaluation functions. Changing the scenario involves recompilation of the binary. It is not possible to write a general purpose application in this case since the customization options of the configuration file are usually not sufficient. To analyze the simulation, either the complete flow field is written out in a VTK file for later post-processing, or results are stored in custom text files or spreadsheets.

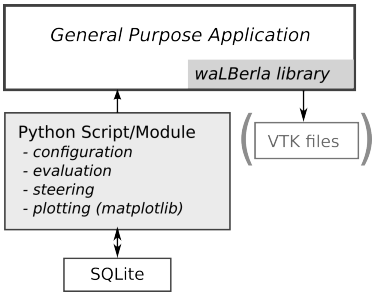

Fig. 7. Typical simulation workflow using Python Interface

Figure 7, in contrast, shows how this workflow is simplified using the Python scripting capabilities of WALBERLA. Whereas previously the description of how to set up and evaluate one scenario was spread out over many files, all this information is now located in a single Python file, or in complex cases, in a Python module. Results can already be evaluated during simulation runs, extracting only the quantities of interest. These reduced results are stored in a relation database, typically using SQLite due to its low configuration overhead. Of course the complete simulation data can still be written out to VTK files but in many cases this not necessary. Visualization and plotting of the collected results can be implemented in the same script, leading to a compact and reusable collection of all information related to a specific simulation setup.

V. Conclusion

We showed the advantages of coupling the WALBERLA C/C++ framework to Python, implementing performance critical parts in C/C++ and higher level functionality, like domain setup, simulation control, and evaluation of results, in Python. We simplified and automated the simulation workflow, starting from scenario definition up to plotting of the results. The flexibility and expressive power of Python enables the user to develop code faster compared to C++. It is therefore also suitable for prototyping of new methods or boundary conditions, a task that previously was done using tools like *Matlab*. The Python interface of WALBERLA makes the framework more attractive for domain experts, which typically are not familiar with C/C++ programming.

References

[1] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde, "WaLBerla: HPC software design for computational engineering simulations," *Journal of Computational Science*, vol. 2, no. 2, pp. 105 – 112, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750311000111

[2] waLBerla Framework, "http://walberla.net," Aug 2014.

[3] Y. Qian, D. d'Humieres, and P. Lallemand, "Lattice BGK models for navier-stokes equation," *EPL (Europhysics Letters)*, vol. 17, no. 6, p. 479, 1992. [Online]. Available: http://iopscience.iop.org/0295-5075/17/6/001

[4] I. Ginzburg, F. Verhaeghe, and D. d'Humieres, "Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions," *Communications in computational physics*, vol. 3, no. 2, pp. 427–478, 2008. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/218788

[5] ——, "Study of simple hydrodynamic solutions with the two-relaxation-times lattice Boltzmann scheme," *Communications in computational physics*, vol. 3, no. 3, pp. 519–581, 2008. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/218787

[6] C. Körner, M. Thies, T. Hofmann, N. Thürey, and U. Rüde, "Lattice boltzmann model for free surface flow for modeling foaming," *Journal of Statistical Physics*, vol. 121, no. 1-2, pp. 179–196, 2005.

[7] S. Donath, C. Feichtinger, T. Pohl, J. Götz, and U. Rüde, "Localized parallel algorithm for bubble coalescence in free surface lattice-boltzmann method," in *Euro-Par 2009 Parallel Processing*. Springer, 2009, pp. 735–746.

[8] J. Götz, C. Feichtinger, K. Iglberger, S. Donath, and U. Rüde, "Large scale simulation of fluid structure interaction using Lattice Boltzmann methods and the 'physics engine'," in *Proceedings of the 14th Biennial Computational Techniques and Applications Conference, CTAC-2008*, ser. ANZIAM J., G. N. Mercer and A. J. Roberts, Eds., vol. 50, Oct. 2008, pp. C166–C188, http://anziamj.austms.org.au/ojs/index.php/ANZIAMJ/article/view/1445 [October 29, 2008].

[9] C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde, "A framework for hybrid parallel flow simulations with a trillion cells in complex geometries," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 35.

[10] boost C++ libraries, "http://www.boost.org/," Aug 2014.

[11] boost.python library, "http://www.boost.org/doc/libs/1_56_0/libs/python/," Aug 2014.

[12] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[13] Python Buffer Protocol, "https://docs.python.org/3/c-api/buffer.html," Aug 2014.

[14] M. Junk and D. Kehrwald, "On the relation between lattice variables and physical quantities in lattice boltzmann simulations," *ITWM Report*, 2006.

[15] Python Units Library Pint, "http://pint.readthedocs.org/," Aug 2014.

[16] E. Jones, T. Oliphant, and P. Peterson, "Scipy: Open source scientific tools for python," *http://www. scipy. org/*, 2001.

[17] D. Joyner, O. Čertík, A. Meurer, and B. E. Granger, "Open source computer algebra systems: Sympy," *ACM Communications in Computer Algebra*, vol. 45, no. 3/4, pp. 225–234, 2012.

[18] A. Henderson, J. Ahrens, and C. Law, *The ParaView Guide*. Kitware Clifton Park, NY, 2004.

[19] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 0090–95, 2007.