A Python extension for the massively parallel framework waLBerla

PyHPC at SC 14, November 17th 2014

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

Chair for System Simulation
Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
Outline

- waLBerla Framework
  - Data Structures
  - Communication patterns
  - Application

- Python interface:
  - Motivation: Simulation Setup
  - Data structure export for simulation evaluation/control
  - Challenges of writing a C++/Python interface
The walBerla Framework
waLBerla Framework

- widely applicable Lattice-Boltzmann from Erlangen
- general HPC software framework, originally developed for CFD simulations with Lattice Boltzmann Method (LBM)
- but other algorithms working on structured grid also possible: phase field models, molecular dynamics with linked cell algorithm
- coupling with in-house rigid body physics engine

Turbulent flow ($Re=11000$) around a sphere

Microstructure simulation of ternary, eutectic solidification process (phase field method)
waLBerla Framework

- Written in C++ with Python extensions
- Hybridly parallelized (MPI + OpenMP)
- No data structures growing with number of processes involved
- Scales from laptop to recent petascale machines
- Parallel I/O
- Portable (Compiler/OS)
- Automated tests / CI servers
- Open Source release planned
Block Structured Grids

- structured grid
- domain is decomposed into blocks
- blocks are the container data structure for simulation data (lattice)
- blocks are the basic unit of load balancing
Block Structured Grids

Complex geometry given by surface

Add regular block partitioning

Load balancing

Discard empty blocks

Allocate block data
Block Structured Grids

Domain partitioning of coronary tree dataset
One block per process

512 processes

458,752 processes
Weak scaling

- Weak scaling - TRT kernel (3.34 million cells per core)

2 islands

![Graph showing weak scaling results](image-url)
walBerla Library

- **field**
- **mpi**
- **domain_decomposition**
  - **cuda**
  - **geometry**
  - **communication**
    - **lattice_boltzmann**
    - **phasefield**
    - **pde**

- **Main Goal: Modularity**
- **Decoupling: heavy use of template concepts**
Python Extension

Motivation: Simulation Setup
Simulation Setup

• Example Application: free surface LBM simulation
• Bubbly Flow through channel

![Diagram of bubbly flow through a channel with liquid film and gravity direction](image)
Simulation Setup

- Example Application: free surface LBM simulation
- Bubbly Flow through channel
Simulation Workflow

Config File
JSON-like format

Voxel File
geometry/boundaries

Scenario Specific Application
with custom steering and
evaluation code

waLBerla library

VTK files

Text Files / Spreadsheet
Result of evaluation routines

Custom Plotting
Motivation: Configuration files

- waLBerla provides a hierarchic key-value pair configuration
- specified in a JSON-like syntax

```plaintext
DomainSetup {
    dx 0.01;
    cells < 200,200,1000>;
}

Physics {
    LBM_Model {
        type SRT;
        relaxation_time 1.9;
    }
}

Geometry {
    free_surface {
        sphere { position < 100,100,500>; radius 10; } //...
    }
    at_boundary { position west; type inflow; vel 0.01; }
}
```

Compute from dx, dt, viscosity?

Use previous values? cells/2”

Check valid range? Physical Units?
Motivation: Configuration files

- Special C++ modules were developed to enhance the config file
  - unit conversion
  - automatic calculation of missing parameters
  - check for valid parameter combinations and ranges
  - support for functions/macros?
- essentially we started to develop a custom scripting language

Python was chosen because:
- popular in scientific community
- easy to use together with C/C++
- boost::python
A Python Extension for the massively parallel framework waLBerla - PyHPC 14
Martin Bauer - Chair for System Simulation, FAU Erlangen-Nürnberg – November 17, 2014

Library Stack

- Interface to expose waLBerla datatypes and call Python functions
- C++ Interface
- C Interface
Callbacks

- Simulation (still) driven by C++ code,
- Callbacks to Python functions
- configuration provided in a Python dict

```python
import waLBerla

@waLBerla.callback( "config" )
def make_config( **kwargs ):
    return { 'DomainSetup' : { 'cells' : (200,200,1000), },
             'Geometry' : { } }

python_coupling::Callback cb ( "config" );
cb(); // run python function
auto config = convertDictToConfig ( cb.returnValue() )
```
Simulation Setup

```python
@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,
        # ...
    }
}
computeDerivedParameters(c)
c['Physical']['dt'] = findOptimalDt(c)
nondimensionalize(c)
return c
```
Simulation Setup

• Further callbacks, for example for boundary setup:

```python
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 }
```
Python Extension

Simulation Evaluation
Simulation Workflow

- **Config File**: JSON-like format
  - **Voxel File**: geometry/boundaries

**Scenario Specific Application**
with custom steering and evaluation code

**walBerla library**

- **VTK files**
  - **Text Files / Spreadsheet**: Result of evaluation routines

- **Custom Plotting**
Callbacks

- Next step: make C++ data structures available in Python functions
- Most of them straightforward, using *boost::python* wrappers

```python
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']

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

- **Field** is the central data structure in waLBerla (stores all simulation data)
  - four dimensional array, that stores all simulation data
  - easy switching between AoS and SoA memory layout
  - distributed: field only represents the locally stored part of the lattice
Field access in Python

- **Field** is the central data structure in waLBerla (stores all simulation data)
  - four dimensional array, that stores all simulation data
  - easy switching between AoS and SoA memory layout
  - distributed: field only represents the locally stored part of the lattice

- user friendly and efficient access from Python is essential
  - no copying
  - should behave like a numpy.array (indexing, slicing, etc.)
Field access in Python

Solution: **Buffer Protocol**

- raw memory is the interface: pointer to beginning together with strides for each dimension
- not wrapped in *boost::python* -> directly use C interface

```python
import waLBerla

def to_numpy( f ):
    return np.asarray( f.buffer() )

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

- heavy use of template code due to performance reasons
- all templates have to instantiated
- which combinations to instantiate?

```cpp
namespace field { // field module
  template<typename T> class Field;
  template<typename T> class FlagField;
}
namespace lbm { // lbm module
  template<typename FieldType> class VelocityAdaptor;
}
namespace vtk { // vtk module
  template<typename FieldType>
  void write(const FieldType & f);
}
namespace postprocessing { // postprocessing module
  template<typename FieldType>
  void generateIsoSurface(const FieldType & f);
}
```
Exporting template code

```cpp
using boost::mpl::vector;
namespace field { // field module
    template<typename T> class;
    template<typename T> class FFieldlagField;

    typedef vector<Field<double>,
        Field<int>,
        FlagField<unsigned char> > FieldTypes;
}
namespace lbm { // lbm module
    template<typename FieldType> class VelocityAdaptor;
    typedef boost::mpl::vector<VelocityAdaptor<double> FieldTypes;
}

typedef boost::mpl::joint_view<
    field::FieldTypes,
    lbm::FieldTypes > MyFieldTypes;

vtk:: python::fieldFunctionExport<MyFieldTypes>;
postprocessing::python::fieldFunctionExport<MyFieldTypes>;
```
• Example: determine maximal velocity in channel
• two stage process, due to distributed storage of lattice

```python
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 = to_numpy(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)
```
• for smaller simulations: simpler (non-distributed) version

```python
import numpy as np

@waLBerla.callback( "at_end_of_timestep" )
def evaluation(blockstorage, bubbles):
    # 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)
```
Evaluation

• Python ecosystem for evaluation
  • *scipy*: FFT to get vertex shedding frequency
  • *sqlite3*: store results of parameter studies
  • *matplotlib*: instantly plot results for singlenode, debugging runs

• Simulation Control: use results of evaluation during simulation
  • *Example*: when velocity does not change in channel a steady flow developed and simulation can be stopped

• All scenario related code (setup, analysis, control) in a single file
Summary
Summary

Config File
JSON-like format

Voxel File
geometry/boundaries

Scenario Specific Application
with custom steering and
evaluation code

waLBerla library

VTK files

Text Files / Spreadsheet
Result of evaluation routines

Custom Plotting

General Purpose Application

waLBerla library

Python Script/Module
- configuration
- evaluation
- steering
- plotting (matplotlib)

SQLite

VTK files

A Python Extension for the massively parallel framework waLBerla - PyHPC 14
Martin Bauer - Chair for System Simulation, FAU Erlangen-Nürnberg – November 17, 2014
Summary

- Python simplifies all stages of the simulation workflow
  - *setup*: nondimensionalization, geometry setup
  - *evaluation*: simulation data accessible as numpy array, storage to database, plotting
  - *control*: definition of stopping criteria, output control
- all scenario dependent code in one file, no custom C++ application necessary any more

Outlook

- add possibility to drive simulation from Python (export all required data structures)
Thank you!