waLBerla: Developing a Massively Parallel HPC Framework

SIAM CS&E 2013, Boston

February 26, 2013

Florian Schornbaum*, Christian Godenschwager*, Martin Bauer*, Matthias Markl†, Ulrich Rüde*

*Chair for System Simulation, †Chair of Metals Science and Technology
Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
Outline

• Introduction

• Implementation
  • From Polymorphism to Concepts
  • Code Generation
  • C++11

• Tools & Maintenance
  • CMake, CTest, and CDash
  • git – Fast, Distributed Source Code Management

• Results / Benchmarks

• Conclusion
Introduction

What is waLBerla?
What are its goals as a software framework?
Introduction

• waLBerla (widely applicable Lattice Boltzmann framework from Erlangen):
  • originally developed for CFD simulations based on the Lattice Boltzmann method (LBM)
  • general HPC software framework (still strong focus on LBM, but multigrid methods also possible)
  • coupling with in-house rigid body physics engine pe
  • written in C++

• Lattice Boltzmann method:
  • stream (neighbors) & collide (cell-local)
  • different models & methods: D2Q9/D3Q19/D3Q27 & SRT/TRT/MRT
Examples

blood flow through coronary arteries
(C. Godenschwager)

turbulent flow (Re=11000) around sphere
(E. Fattahi, D. Weingaertner)
Examples

electron beam melting metal powder: 3D printing (R. Ammer, M. Markl)

[ liquid: waLBerla (LBM) metal powder: pe (fully resolved rigid bodies) ]

POV-Ray rendering of actual simulation
HPC Framework Goals

• High single node performance
  • vectorized compute kernels (SIMD: SSE, AVX, QPX)
  • performance modeling

• Scalability
  • run with $10^0$ to $10^6$ cores/processes ("unlimited" number of processes)
  • only fully distributed data structures/algorithms and local, process-to-process (no all-to-one or all-to-all) communication
HPC Framework Goals

• Flexibility / Extensibility
  • arbitrary number of different compute kernels
  • interchangeable/configurable boundary handling, communication, and input/output mechanisms
  ⇒ allow any arbitrary combination of “functionality”

• Support for many platforms/compilers
  • laptops, desktops, supercomputers (Linux/Unix, Windows)
  • compilers: GCC, Intel, Visual Studio, IBM
  • parallelization: no MPI, pure MPI, pure OpenMP, MPI+OpenMP
Implementation

Polymorphism, concepts, or code generation? Where to with C++?
Data Structures / Software Design

• Domain decomposition
  • blocks containing uniform grids

uniform block decomposition
non-uniform, octree-like block decomposition
non-uniform, octree-like block decomposition

• Former software design mistake:
  • tight coupling between one particular domain decomposition and “everything” else (compute kernels, input, output, etc.)
Better design: make use of **polymorphism**

- similarities between different block decompositions
- introduce interfaces/base classes for block structures, blocks, and block IDs
- a lot of the framework functionality can already be implemented in these base classes
- many algorithms can be implemented using these interfaces
  ⇒ much better maintainability
  ⇒ high reusability of existing code (input, output, ...)

Performance?

- overhead of virtual function calls can be neglected for high-level framework functionality
• Interchangeability / reusability of functionality in low-level, performance-critical parts of code?
  • example: boundary handling (different boundary conditions)
  • polymorphism: works, but too much overhead
  • solution: combination of templates (compile time evaluation) and “concepts”

• Concept:
  • “interface” without polymorphism – no language support (considered for C++11, removed from draft in 2009)
  • “[...] is a description of supported operations on a type, including syntax and semantics [...] related to abstract base classes but [...] do not require a subtype relationship” (Wikipedia)
Data Structures / Software Design

• Example:

```cpp
template< typename T >
int foo( const T& t, const int bar ) { return t.baz(bar); }
```

⇒ every type usable with function “foo” must have a public member function “int baz( const int )”

• Concepts and boundary handling:
  • boundary condition: must implement a specific concept
  • boundary handling: receives an arbitrary number of boundary conditions as templates
• Concepts and boundary handling (cond.):
  • At compile time, an application-specific boundary handling is assembled from a list of boundary conditions.

⇒ high performance (identical to hard-coded, problem-adapted function)
⇒ highly customizable

• Conclusion:
  • high-level framework functionality: polymorphism
  • low-level, performance-critical code: concepts & templates
Data Structures / Software Design

• Interchangeability / reusability of functionality in low-level, performance-critical parts of code (cond.):
  • example: The same data structures and the same algorithms can be used with different stencils.
    
    D2Q9:  
    
    D2Q5:  

• define concept “stencil”, use templates, write code once
  ⇒ flexible, reusable, high performance ...

• .... but: hard to keep consistent when concept “stencil” is changed (→ maintenance overhead, prone to mistakes)

• However, all stencils are constructed using the same pattern!
  (different to boundary handling & boundary conditions)
• Concept “stencil” (cond.):
  ⇒ All stencils are constructed following the same pattern!
  • use template metaprogramming to generate all required stencils at compile time? → failed attempt
    • messy, complex C++ template code
    • C++03: no guarantees that certain specific values are evaluated at compile time (C++11: constexpr!)
  ⇒ code generation via **python** scripts:
    • file “template” used to generate all stencils
    • maintainability: no code duplication
    • compile time evaluation (concepts & templates)
  ⇒ high performance
C++11 / Software Design

• C++11: Embrace the full capabilities of your programming language!

• Bad news first:
  • The common subset of features implemented by all major compilers (GCC, Intel, IBM, Visual Studio) is very small.
  • Even if (many) language features are already implemented, often compiling the standard library (and/or boost) will fail when C++11 is fully activated.
  • cannot use: variadic templates, constexpr, lambdas, ...
C++11 / Software Design

- C++11: Embrace the full capabilities of your programming language!

- The good news (features you can use):
  - compile time checks (with full support by the language):
    
    ```cpp
    static_assert( sizeof(T) >= 8, "some meaningful message" );
    ```
  - "auto" (much less typedefs/screen filling types):
    
    ```cpp
    for( auto block = blockStorage.begin();
        block != blockStorage.end(); ++block )
    {
      const auto& aabb = block->getAABB();
      [...]
    }
    ```

  ⇒ makes code a lot cleaner and faster to develop
Tools & Maintenance

Can there be too much Kitware?
Distribution trumps centralization?
Building and Testing

• walBerla uses **CMake**

• CMake: platform-/compiler-independent build system
  • Makefile on Linux/Unix, project files for Visual Studio, ... 
  • compiler support: GCC, Intel, IBM, Visual Studio, ... 
  • available on supercomputers (JUQUEEN (Top5), SuperMUC (Top6))

⇒ CMake allows walBerla to be build anywhere from a MacBook Air to Blue Gene/Q supercomputers with just one CMake configuration/setup.
Building and Testing

- Testing: CTest
  - fully integrated into CMake ecosystem
  - tests can be manually executed, but become particularly interesting when combined with our own ...

- ... fully automatic build and testing environment
  - controlled by **Python build scripts** (rely on XML configuration files)
  - triggered by every check-in in main repository
  - several different combinations of compilers and build types: GCC or Intel, MPI or no MPI, Release or Debug, ...
  - nightly builds: Windows & Visual Studio
  - reports results to CDash ⇒
### Building and Testing

- waLBerla uses **CDash**

#### Nightly

<table>
<thead>
<tr>
<th>Site</th>
<th>Build Name</th>
<th>Update</th>
<th>Configure</th>
<th>Build</th>
<th>Test</th>
<th>Build Time</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Files</td>
<td>Error</td>
<td>Warn</td>
<td>Error</td>
<td>Warn</td>
<td>Not Run</td>
</tr>
<tr>
<td>I10STAFF25</td>
<td>Win32-vs10 [trunk]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20.14</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Git Buildscript

<table>
<thead>
<tr>
<th>Site</th>
<th>Build Name</th>
<th>Update</th>
<th>Configure</th>
<th>Build</th>
<th>Test</th>
<th>Build Time</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Files</td>
<td>Error</td>
<td>Warn</td>
<td>Error</td>
<td>Warn</td>
<td>Not Run</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master intel Release NoMPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master intel Release MPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master intel Debug NoMPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master intel Debug MPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master gcc Release NoMPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master gcc Release MPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master gcc Debug NoMPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i10test6</td>
<td>dea6519 master gcc Debug MPI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
waLBerla: Developing a Massively Parallel HPC Framework

Florian Schornbaum, FAU Erlangen-Nürnberg

Revision Control

• waLBerla used to use Subversion (SVN)

• waLBerla now uses 🎯 git
  • distributed revision control system
  • git & supercomputers:
    • available on all supercomputers we are currently using
    • git just works:
      • allows for pull/fetch and push operations
      • Even if outgoing connections are blocked, you can still push (via ssh) your local commits to your repository on the cluster.
    • unbelievably fast!
Results / Benchmarks

Blue Gene/Q or Xeon? IBM vs. Intel!
• Testing environments:
  - JUQUEEN (TOP500: 5), Blue Gene/Q, 459K cores, 1 GB/core
    - IBM compiler 12.1, IBM MPI
    - Green500: 5 (2,102.12 MFLOPS / Watt)
  - SuperMUC (TOP500: 6), Intel Xeon, 147K cores, 2 GB/core
    - Intel compiler 12.1.6, IBM MPI
    - world's fastest x86-based supercomputer

• Benchmark:
  - LBM fluid simulation (D3Q19)
  - uniform grid (lid driven cavity, flow around object in channel)
  - weak scaling (const. number of cells per core)
  - MLUP/s: million/mega lattice cell updates per second
Results / Benchmarks

• SuperMUC – single node (3.34 million cells per core)

⇒ limited by memory bandwidth

LBM compute kernel type

more complex, but also much more relevant for real simulations!
Results / Benchmarks

- SuperMUC – single node (3.34 million cells per core)

⇒ limited by memory bandwidth

already quite optimized!

naïve, straightforward implementation

MLUP/s

cores

0 20 40 60 80 100 120 140 160 180

1 2 4 8 12 16

SRT

TRT

naïve, straightforward implementation
Results / Benchmarks

- JUQUEEN – single node (1.73 million cells per core)

\[ \Rightarrow \text{limited by memory bandwidth} \]

already quite optimized!

 naïve, straightforward implementation
Results / Benchmarks

- JUQUEEN – single node (1.73 million cells per core)

⇒ limited by memory bandwidth

hybrid version
(4 threads per core)
Results / Benchmarks

- SuperMUC – TRT kernel (3.34 million cells per core)

jobs are already in the queue, waiting to be executed ...
Results / Benchmarks

- JUQUEEN – TRT kernel (1.73 million cells per core)

1.68 x 10^{12} cells updated per second!
(19 values per cell)

⇒ 0.33 PFlop/s (0.33 x 10^{15} Flop/s)
Summary & Conclusion

So, what was it all about?
What next?
Summary & Conclusion

• Usability of the framework & reusability of code: possible, even if main focus is on speed/performance

• Programming language of choice: C++ (C++11)

• Maintainability:
An automatic build and testing system allows waLBerla to support a wide variety of platforms (Linux, Windows, laptops, desktops, supercomputers, MPI, no MPI, OpenMP, hybrid, ...) and compilers (GCC, Intel, IBM, Visual Studio).

future plan for waLBerla: go open source
(current owner? which license? which parts of the code?)
THANK YOU FOR YOUR ATTENTION!

QUESTIONS?