waLBerla: Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation

SIAM Parallel Processing for Scientific Computing 2012

February 16, 2012

Florian Schornbaum, Christian Feichtinger, Harald Köstler, Ulrich Rüde

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

• Introduction
• Motivation / Problem Description
  – Current Framework Capabilities
  – Future Goals & Extensions
• Prototyping Environment
  – Implementation
  – Data Structures
  – Distributed Refinement/Coarsening Algorithm
  – Procedure Virtualization / Virtual Blocks
  – Load Balancing
• Results / Benchmarks
• Summary & Conclusion
WaLBerla: Minimize hardware and software costs

Applications Free-Surface Flows Particulate Flows Hardware

Performance Parallelization Efficient Kernels Performance Modeling Software Engineering

Sweeps Patches & Blocks Meta Data
WaLBerla: Patch concept

Simulation Domain

Patch

Unknown / Cell

Block

Block Info:
- Application
- Rank
- Is Allocated
- AABB
- BlockID

Block Data:
- Simulation Data:
  - Cartesian Data
  - Unstructured Data
- Configurable:
  - Data Structures
  - Algorithms
  - Optimizations
• **waLBerla:**
  A massively parallel software framework originally developed for **CFD simulations** based on the Lattice Boltzmann method (LBM)

• **Lattice Boltzmann method:**
  In every time step, each cell in a discretized simulation space exchanges information with its directly adjacent neighbors:

  → **high data locality**
  → especially well suited for extensive **parallelization**
Currently, the waLBerla framework does not support refinement. → The simulation space is always **regularly discretized**.

For parallel simulations, each process is assigned agglomerates of several thousands of cells ("blocks" of cells). → geometric distribution
• The required inter- and intra-process communication schemes are relatively easy to understand and to implement.
  → Data must be exchanged only between neighboring blocks.
  → straight-forward parallelization of large simulations
• waLBerla will be extended to support **grid refinement** (for more information on grid refinement & LBM see Filippova et al., Dupuis et al., Krafczyk et al.).

• restrictions for and consequences of grid refinement:
  – 2:1 size ratio of neighboring cells

  ![Diagram showing grid refinement](image)

  higher resolution in areas covered with obstacles

  → With the Lattice Boltzmann method, on the fine grid, **twice as many time steps** need to be performed as on the coarse grid.
walBerla will be extended to support **grid refinement** (for more information on grid refinement & LBM see Filippova et al., Dupuis et al., Krafczyk et al.).

**restrictions for and consequences of grid refinement:**

- 2:1 size ratio of neighboring cells

With the Lattice Boltzmann method, on the fine grid, twice as many time steps need to be performed as on the coarse grid.

higher resolution in areas covered with obstacles
• walBerla will be extended to support **grid refinement** (for more information on grid refinement & LBM see Filippova et al., Dupuis et al., Krafczyk et al.).

• restrictions for and consequences of grid refinement:
  – **2:1 size ratio of neighboring cells**

  ![Diagram](image)

  higher resolution in areas covered with obstacles

→ With the Lattice Boltzmann method, on the fine grid, twice as many time steps need to be performed as on the coarse grid.
wallBerla will be extended to support **grid refinement** (for more information on grid refinement & LBM see Filippova et al., Dupuis et al., Krafczyk et al.).

- restrictions for and consequences of grid refinement:
  - 2:1 size ratio of neighboring cells

  ![Diagram showing grid refinement](image)

  higher resolution in areas covered with obstacles

  → With the Lattice Boltzmann method, on the fine grid, **twice as many time steps** need to be performed as on the coarse grid.
Motivation / Problem Description
Future Goals & Extensions

- restrictions for and consequences of grid refinement (cont.):
  - In 3D, one refinement step leads to eight times as many cells being required in the refined area:
    \[
    \rightarrow \text{memory consumption } \times 8 \quad \& \quad \text{generated workload } \times 16
    \]
  - If more than one refinement level is used, the 2:1 size ratio of neighboring cells must be obeyed:
    \[
    \begin{array}{c}
    \begin{array}{c}
    \begin{array}{c}
    \begin{array}{c}
    \begin{array}{c}
    \end{array}
    \end{array}
    \end{array}
    \end{array}
    \end{array}
    \rightarrow
    \begin{array}{c}
    \begin{array}{c}
    \begin{array}{c}
    \begin{array}{c}
    \end{array}
    \end{array}
    \end{array}
    \end{array}
    \]
  - If \( n \) refinement levels are used, then
    \[
    \frac{\text{memory workload on the finest grid}}{\text{workload on the coarsest grid}} = \frac{8^{n-1} \times \text{memory}}{16^{n-1} \times \text{workload}}
    \]
Motivation / Problem Description

Future Goals & Extensions

• restrictions for and consequences of grid refinement (cont.):
  – In 3D, one refinement step leads to eight times as many cells being required in the refined area:
    → memory consumption ×8 & generated workload ×16

  – If more than one refinement level is used, the 2:1 size ratio of neighboring cells must be obeyed:

  – If \( n \) refinement levels are used, then
    \[
    \frac{\text{memory workload on the finest grid}}{\text{workload on the coarsest grid}} = \frac{8^{n-1} \times \text{memory}}{16^{n-1} \times \text{workload}}
    \]
restrictions for and consequences of grid refinement (cont.):

- In 3D, one refinement step leads to eight times as many cells being required in the refined area:
  → memory consumption ×8 & generated workload ×16

- If more than one refinement level is used, the 2:1 size ratio of neighboring cells must be obeyed:

- If \( n \) refinement levels are used, then

\[
\frac{\text{memory workload on the finest grid}}{\text{memory workload on the coarsest grid}} = \frac{8^{n-1} \times \text{memory}}{16^{n-1} \times \text{workload}}
\]
Motivation / Problem Description
Future Goals & Extensions

• restrictions for and consequences of grid refinement (cont.):
  – In 3D, one refinement step leads to eight times as many cells being required in the refined area:
    → memory consumption \times 8 & generated workload \times 16
  – If more than one refinement level is used, the 2:1 size ratio of neighboring cells must be obeyed:

  ![Diagram showing grid refinement](image)

  – If \( n \) refinement levels are used, then

\[
\text{memory workload on the finest grid} = \frac{3^{n-1}}{16^{n-1}} \times \text{memory workload on the coarsest grid}
\]
Motivation / Problem Description
Future Goals & Extensions

- restrictions for and consequences of grid refinement (cont.):
  - In 3D, one refinement step leads to eight times as many cells being required in the refined area:
    → memory consumption $\times 8$ & generated workload $\times 16$
  - If more than one refinement level is used, the 2:1 size ratio of neighboring cells must be obeyed:

```
\[ \text{memory workload on the finest grid} = \frac{8^{n-1}}{16^{n-1}} \times \text{memory workload on the coarsest grid} \]
```
Motivation / Problem Description
Future Goals & Extensions

• In order to achieve good load balancing, subdividing the simulation space into **equally sized regions won’t work**.

→ Each process must be assigned the same amount of work (the workload is given by the number of cells weighted by the number of time steps that need to be performed on the corresponding grid level).

→ Not trivial to solve for billions of cells!
In order to achieve good load balancing, subdividing the simulation space into equally sized regions won’t work.

Each process must be assigned the same amount of work (the workload is given by the number of cells weighted by the number of time steps that need to be performed on the corresponding grid level). Not trivial to solve for billions of cells!
In order to achieve good load balancing, subdividing the simulation space into **equally sized regions won’t work**.

Each process must be assigned the same amount of work (the work-load is given by the **number of cells weighted by the number of time steps** that need to be performed on the corresponding grid level).

→ Not trivial to solve for billions of cells!
In order to achieve good load balancing, subdividing the simulation space into equally sized regions won’t work. Each process must be assigned the same amount of work (the workload is given by the number of cells weighted by the number of time steps that need to be performed on the corresponding grid level).

Not trivial to solve for billions of cells!
• The problem gets even worse if the **fine regions** are not static but **dynamically change their locations** (moving obstacles etc.).
  
  → Areas initially consisting of coarse cells will require much more memory and generate a lot more workload after being refined (and vice versa).
  
  → massive workload & memory fluctuations!

• Performing **global** refinement, coarsening, and load balancing (by synchronizing all processes or using a master-slave scheme) can be extremely expensive or maybe even impossible for simulations with billions of cells distributed to thousands of processes.

  → solution: fully **distributed algorithms** working in parallel
• The problem gets even worse if the **fine regions** are not static but **dynamically change their locations** (moving obstacles etc.).
  → Areas initially consisting of coarse cells will require much more memory and generate a lot more workload after being refined (and vice versa).
  ⇒ **massive workload & memory fluctuations** !

• Performing **global refinement, coarsening, and load balancing** (by synchronizing all processes or using a master-slave scheme) can be extremely expensive or maybe even impossible for simulations with billions of cells distributed to thousands of processes.
  → solution: fully **distributed algorithms** working in parallel
• The problem gets even worse if the **fine regions** are not static but **dynamically change their locations** (moving obstacles etc.).
  → Areas initially consisting of coarse cells will require much more memory and generate a lot more workload after being refined (and vice versa).
  ⇒ **massive workload & memory fluctuations** !

• Performing **global** refinement, coarsening, and load balancing (by synchronizing all processes or using a master-slave scheme) can be extremely **expensive or maybe even impossible** for simulations with billions of cells distributed to thousands of processes.

  → solution: fully distributed algorithms working in parallel
The problem gets even worse if the **fine regions** are not static but **dynamically change their locations** (moving obstacles etc.).

→ Areas initially consisting of coarse cells will require much more memory and generate a lot more workload after being refined (and vice versa).

⇒ **massive workload & memory fluctuations**!

• Performing **global** refinement, coarsening, and load balancing (by synchronizing all processes or using a master-slave scheme) can be extremely **expensive or maybe even impossible** for simulations with billions of cells distributed to thousands of processes.

→ solution: **fully distributed algorithms** working in parallel

Motivation / Problem Description
Future Goals & Extensions
Outlines:

- Introduction
- Motivation / Problem Description
  - Current Framework Capabilities
  - Future Goals & Extensions
- Prototyping Environment
  - Implementation
  - Data Structures
  - Distributed Refinement/Coarsening Algorithm
  - Procedure Virtualization / Virtual Blocks
  - Load Balancing
- Results / Benchmarks
- Summary & Conclusion
In order to be able to deal with all of these problems, **new and adapted data structures and algorithms** are required.

A **prototyping environment** has been created within the waLBerla framework that solely focuses on the development of these new data structures and distributed algorithms.

- No actual Lattice Boltzmann fluid simulation is executed.
- All the data that is required for the LBM only exists in form of accumulated, abstract information regarding workload and memory.
- Adaptive refinement is simulated by moving spherical objects through the simulation and demanding a fine resolution around these objects.

The prototyping environment allows for a **fast and efficient development and testing of different concepts and structures**.
The prototyping environment (written in C++) is not parallelized with MPI but only with **OpenMP**. It runs on **shared memory systems**.

Thousands of processes running in parallel using distributed algorithms for refinement and balancing are only simulated.

**Advantages:**

- Fast development and testing (→ thousands of processes can be **simulated on a desktop computer**)
- All tasks are also solved with easy to understand, global algorithms which are then used to **validate the results of the fully distributed, parallel algorithms**.
• Algorithms working on a cell-based structure cannot be implemented efficiently.

→ highly irregularly shaped partitions of the simulation domain
→ completely irregular communication schemes
→ Computation sweeps over blocks of cells resulting from the current homogenous discretization are much more efficient.

⇒ The new structure is also based on blocks of cells (e.g., 40×40×40). (All cells in one block are of the same size.)
• Algorithms working on a cell-based structure cannot be implemented efficiently.

→ highly irregularly shaped partitions of the simulation domain
→ completely irregular communication schemes
→ Computation sweeps over blocks of cells resulting from the current homogenous discretization are much more efficient.

⇒ The new structure is also based on blocks of cells (e.g., 40×40×40). (All cells in one block are of the same size.)
• Algorithms working on a cell-based structure cannot be implemented efficiently.

→ highly irregularly shaped partitions of the simulation domain
→ completely irregular communication schemes
→ Computation sweeps over blocks of cells resulting from the current homogenous discretization are much more efficient.

⇒ The new structure is also based on blocks of cells (e.g., 40×40×40). (All cells in one block are of the same size.)
• Algorithms working on a cell-based structure cannot be implemented efficiently.

⇒ highly irregularly shaped partitions of the simulation domain
⇒ completely irregular communication schemes
⇒ Computation sweeps over blocks of cells resulting from the current homogenous discretization are much more efficient.

⇒ The new structure is also based on blocks of cells (e.g., 40×40×40). (All cells in one block are of the same size.)
• Algorithms working on a cell-based structure cannot be implemented efficiently.

→ **highly irregularly shaped partitions** of the simulation domain
→ completely **irregular communication schemes**
→ **Computation sweeps** over blocks of cells resulting from the current homogenous discretization are much more efficient.

⇒ The new structure is also based on **blocks of cells** (e.g., 40×40×40).
(All cells in one block are of the same size.)
Prototyping Environment
Data Structures

• The **2:1 cell size ratio restriction** causes two neighboring blocks to have the same cell size or to differ by only one refinement level.

region in the simulation domain where the underlying application demands a fine resolution

• What makes this structure special/different:
  No concepts and structures typically associated with trees (father-child connections, inner nodes, etc.) are used. Each block only knows all of its direct neighbors → perfect for parallelization!
The **2:1 cell size ratio restriction** causes two neighboring blocks to have the same cell size or to differ by only one refinement level.

region in the simulation domain where the underlying application demands a fine resolution

- What makes this structure special/different:
  No concepts and structures typically associated with trees (father-child connections, inner nodes, etc.) are used. Each block only knows all of its direct neighbors → perfect for parallelization!
• The **2:1 cell size ratio restriction** causes two neighboring blocks to have the same cell size or to differ by only one refinement level.

geometrically:

**forest of octrees**

(blocks = leaves)

region in the simulation domain where the underlying application demands a fine resolution

• What makes this structure special/different:

No concepts and structures typically associated with trees (father-child connections, inner nodes, etc.) are used. Each block only knows all of its direct neighbors → perfect for parallelization!
The **2:1 cell size ratio restriction** causes two neighboring blocks to have the same cell size or to differ by only one refinement level.

**Prototyping Environment**

**Data Structures**

- geometrically: **forest of octrees** (blocks = leaves)
- region in the simulation domain where the underlying application demands a fine resolution

**What makes this structure special/different:**
No concepts and structures typically associated with trees (father-child connections, inner nodes, etc.) are used. **Each block only knows all of its direct neighbors** → perfect for parallelization!
The 2:1 cell size ratio restriction causes two neighboring blocks to have the same cell size or to differ by only one refinement level.

geometrically:
forest of octrees
(blocks = leaves)

region in the simulation domain where the underlying application demands a fine resolution

What makes this structure special/different:
No concepts and structures typically associated with trees (father-child connections, inner nodes, etc.) are used. Each block only knows all of its direct neighbors → perfect for parallelization!
If the area that requires the finest resolution changes, the data structure must be adapted accordingly:

From now on, each box represents an entire block of cells.

If one block is refined, more additional blocks may be affected:
If the area that requires the finest resolution changes, the data structure must be adapted accordingly:

From now on, each box represents an entire block of cells.

If one block is refined, more **additional blocks may be affected**:
• If the area that requires the finest resolution changes, the data structure must be adapted accordingly:

From now on, each box represents an entire block of cells.

• If one block is refined, more additional blocks may be affected:
• If the area that requires the finest resolution changes, the data structure must be adapted accordingly:

From now on, each box represents an entire block of cells.

• If one block is refined, more additional blocks may be affected:
The same holds true if multiple blocks are reunited to one single block (→ coarsening):

Refinement & coarsening is performed in parallel by a fully distributed algorithm.

The runtime of these algorithms only depends on the number of grid levels, not the number of processes!
• The same holds true if multiple blocks are reunited to one single block (coarsening):

• Refinement & coarsening is performed in parallel by a fully distributed algorithm.

→ The runtime of these algorithms only depends on the number of grid levels, not the number of processes!
• The same holds true if multiple blocks are reunited to one single block (→ coarsening):

• Refinement & coarsening is performed in parallel by a fully distributed algorithm.

→ The runtime of these algorithms only depends on the number of grid levels, not the number of processes!
• The same holds true if multiple blocks are reunited to one single block (→ coarsening):

• Refinement & coarsening is **performed in parallel by a fully distributed algorithm**.

  → The runtime of these algorithms only depends on the number of grid levels, not the number of processes!
• Idea: Each block creates a **virtual representation** of itself:
  
  – Each virtual block has a **very small memory footprint** (no cells but only values like 'workload' and 'memory size' are stored).
  
  – All algorithms (refinement, coarsening, and load balancing) operate on these virtual blocks.
    → If a block moves from one process to another, only a **small amount of memory must be communicated**.
  
  – Only at the end of the refinement-coarsening-balancing pipeline the actual blocks follow their virtual blocks to the designated target processes (and only then refinement and coarsening is performed on the actual cells).
Starting situation:

1. Initialization:

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
Prototyping Environment
Procedure Virtualization / Virtual Blocks

Starting situation:

1. Initialization:

- Blocks may be aggregated
- Block needs to be refined

process distribution

actual blocks

virtual blocks
Starting situation:

process distribution

2. Refinement:

blocks may be aggregated
block needs to be refined
Starting situation:

2. Refinement:
Prototyping Environment
Procedure Virtualization / Virtual Blocks

Starting situation:

process distribution

2. Refinement:

blocks may be aggregated
block needs to be refined

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
Starting situation:

process distribution

blocks may be aggregated
block needs to be refined

3. Coarsening:
Starting situation:

- Process distribution

3. Coarsening:

- Blocks may be aggregated
- Block needs to be refined
3. Coarsening:

Starting situation:

process distribution

blocks may be aggregated
block needs to be refined
4. Load Balancing:

Starting situation:

process distribution

- blocks may be aggregated
- block needs to be refined
Starting situation:

process distribution

- blocks may be aggregated
- block needs to be refined

5. Finalization:

actual blocks

virtual blocks
• Each block has the same number of cells (→ identical memory consumption), but smaller cells generate more workload.
  – In a simulation with 5 different grid levels, 2 blocks on the finest level generate the same amount of work than 32 blocks on the coarsest level ...
  – ... yet 32 blocks might not fit into the memory of one process.
• Blocks assigned to the same process should be close.

⇒ Load balancing problem/situation #1:
Some processes may reach their memory limit without generating as much work as the average process.
• **The blocks should be large**, i.e., they should contain many cells:
  → few (maybe only one) blocks per process
  → minimizes communication cost
  → enables efficient computation algorithms

• Only entire blocks can be exchanged between processes:
  → many blocks per process (certainly good for balancing)
  → **The blocks should be small.**

⇒ Load balancing problem/situation #2:

On average, each process owns about 4 to 10 blocks and possesses 20 to 25 neighbors (in 3D).
Implemented static load balancing strategies:

- **Space-filling curves:**
  - **Z-order** (aka Morton order or Morton code)
  - **Hilbert curve**
    - Both curves can be constructed by a depth-first search.

- A custom greedy algorithm which aggregates neighboring blocks

Comparison of these three methods:

- **number of processes:** \( \text{greedy} \leq \text{Hilbert} < \text{Morton} \)
- **partition quality** (intra-process com.): \( \text{greedy} > \text{Hilbert} > \text{Morton} \)
- **runtime** (less is better): \( \text{Morton} = \text{Hilbert} \ll \text{greedy} \)

→ all three algorithms run in \( O(\#\text{processes}) \)
Dynamic load balancing is based on a **diffusive algorithm**:

- The 'work flow' between neighboring processes is calculated.

If the flows on all edges were met exactly, almost perfect load balancing could be achieved.

The flows cannot be met:

- Available/free memory must be taken into account
- Fewer blocks per process than connections to other processes

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation

Harald Köstler, FAU Erlangen-Nürnberg
Dynamic load balancing is based on a **diffusive algorithm**:

- The '**work flow**' between neighboring processes is calculated.

If the flows on all edges were met exactly, almost perfect load balancing could be achieved.

The flows cannot be met:

- Available/free memory must be taken into account
- Fewer blocks per process than connections to other processes

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation

Harald Köstler, FAU Erlangen-Nürnberg
Dynamic load balancing is based on a **diffusive algorithm**:
- The 'work flow' between neighboring processes is calculated.

If the flows on all edges were met exactly, almost perfect load balancing could be achieved.

The flows cannot be met:
- Available/free memory must be taken into account
- Fewer blocks per process than connections to other processes

one process with 5 blocks, workload per block and work flow per edge (process graph) are illustrated
Dynamic load balancing is based on a **diffusive algorithm**:

- The *work flow* between neighboring processes is calculated.

If the flows on all edges were met exactly, almost perfect load balancing could be achieved.

The flows cannot be met:

- Available/free memory must be taken into account
- Fewer blocks per process than connections to other processes

one process with 5 blocks, workload per block and work flow per edge (process graph) are illustrated
Dynamic load balancing is based on a **diffusive algorithm**:  
- The 'work flow' between neighboring processes is calculated.

If the flows on all edges were met exactly, almost perfect load balancing could be achieved.

The flows cannot be met:  
- Available/free memory must be taken into account
- Fewer blocks per process than connections to other processes

one process with 5 blocks, workload per block and work flow per edge (process graph) are illustrated
Dynamic load balancing is based on a **diffusive algorithm**:
- The '**work flow**' between neighboring processes is calculated.

If the flows on all edges were met exactly, almost perfect load balancing could be achieved.

**The flows cannot be met:**
- Available/free memory must be taken into account
- Fewer blocks per process than connections to other processes

one process with 5 blocks, workload per block and work flow per edge (process graph) are illustrated
The basic ideas behind our current implementation:

1) Refinement and coarsening can both lead to too many (virtual) blocks to be located on the same process.
   → By redistributing these blocks, a distributed algorithm makes sure that the memory limit is not violated.

2) The diffusive load balancing algorithm ...
   – ... does not violate the memory limit (receiving processes must always authorize block exchanges)
   – ... uses the calculated work flows for guidance:
     • sum of flow → number of blocks to be sent/received
     • work flow, memory usage of all neighbors, etc. → used for guidance where to send (sending processes decide)
Outline

• Introduction
• Motivation / Problem Description
  – Current Framework Capabilities
  – Future Goals & Extensions
• Prototyping Environment
  – Implementation
  – Data Structures
  – Distributed Refinement/Coarsening Algorithm
  – Procedure Virtualization / Virtual Blocks
  – Load Balancing
• Results / Benchmarks
• Summary & Conclusion
'simulated' simulation: 14 rising bubbles
→ high resolution around these bubbles
14 rising bubbles (→ high resolution around these bubbles)
5 different grid levels – initially: 15 016 blocks (40×40×40 cells)
14 rising bubbles (→ high resolution around these bubbles)
5 different grid levels – initially: 15 016 blocks (40×40×40 cells)
300 processes – initially: 15,016 blocks & 961,024,000 cells

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
300 processes – initially: 15,016 blocks & 961,024,000 cells

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
300 processes – initially: 15,016 blocks & 961,024,000 cells

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
300 processes – initially: 15,016 blocks & 961,024,000 cells
Results / Benchmarks
160 000 Processes – Load Balancing

160 000 pro. – initially: 808 176 blocks & 51 723 264 000 cells

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
160 000 pro. – initially: 808 176 blocks & 51 723 264 000 cells
160 000 Processes – Load Balancing

160 000 pro. – initially: 808 176 blocks & 51 723 264 000 cells

Towards an Adaptive, Dynamically Load-Balanced, Massively Parallel Lattice Boltzmann Fluid Simulation
Harald Köstler, FAU Erlangen-Nürnberg
We have all ingredients required for very large, adaptive, dynamically load balanced Lattice Boltzmann fluid simulations:

- handling of/interpolation between different grid resolutions (→ Filippova et al., Dupuis et al., Krafczyk et al.)
- our contribution: all the necessary data structures and algorithms for performing simulations in **massively parallel environments** (100,000 processes and more)
  - very high data locality within the **fully distributed 'blocks of cells' data structure**
  - manipulation (refinement, balancing, etc.) only through **distributed/diffusive algorithms**

prototyping environment → production code (waLBerla framework)
THE END

Questions?