

Molecular Dynamics - Assignment 5

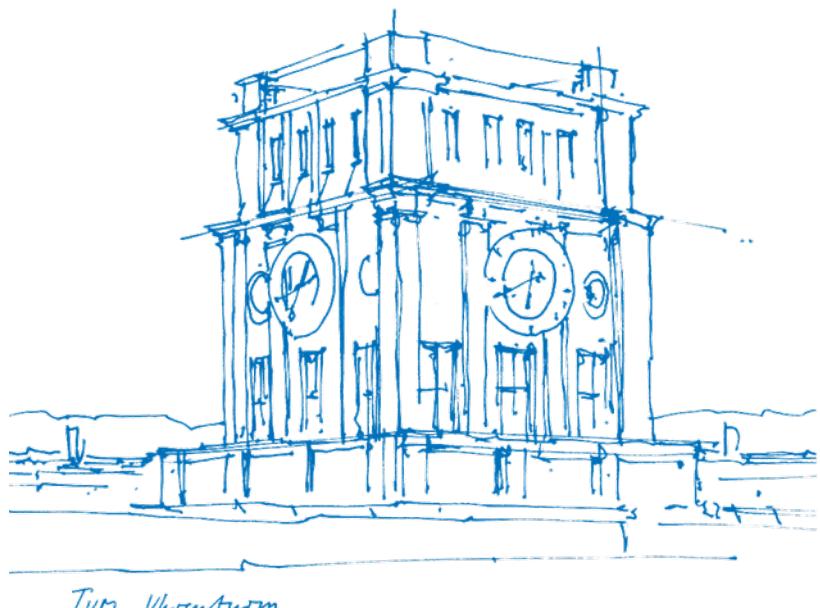
Alex Hocks Jan Hampe Johannes Riemenschneider

Technische Universität München

TUM CIT

Lehrstuhl für wissenschaftliches Rechnen

1. Februar 2023



Implementation of Membranes

- Membrane is wrapper class around 2d-vector of Particle IDs
- Actual Particles still get stored in ParticleContainer

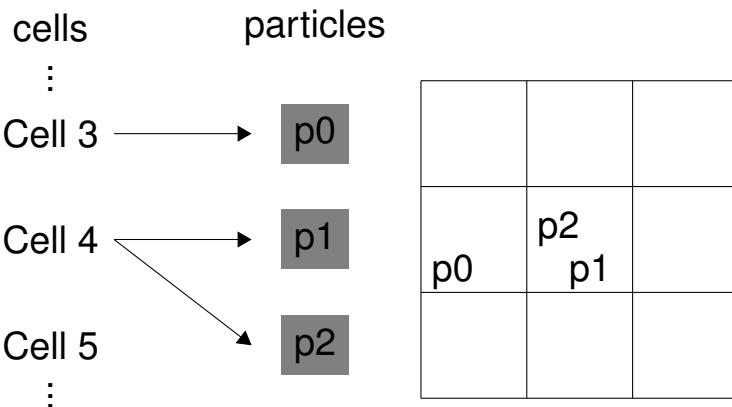
Implementation of immovable Particles

- Set mass of particle to -inf
- refactor code to ignore those particles wherever needed (e.g. Thermostat)

Preparation for Multithreading- Refactoring the Particle Container

Our current implementation of the Cell Data structure (as displayed in Assignment 3):

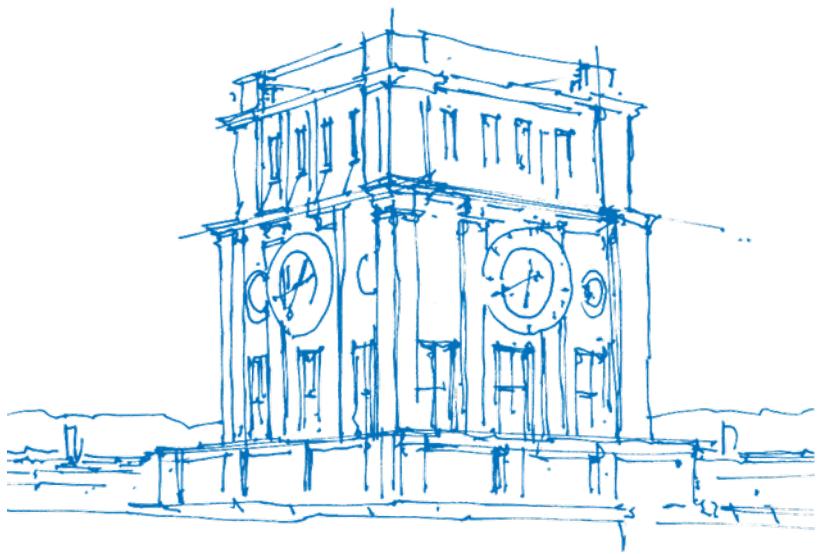
- Particles get stored in one giant vector
- Each Cell keeps references to their particles
- No sorting or copying takes place



Our changes to avoid false sharing:

- Sort particle according to cell index of their respective cell
- Add size of one cacheline between each cell as padding

The Multithreading Odyssey



TUM Uhrenturm

Defining Terminology

Terms necessary to talk about our multithreading approaches efficiently.

- **Task:** A Task is a pair of cells that should interact with each other. Tasks get represented by pairs of cell indices.
- **Task Model:** The task model defines the data structure that these tasks are stored in.
- **Distribution Strategy:** Many task models require splitting up bundles of tasks into buckets. The distribution strategy defines the strategy used for this splitting process.

Distribution Strategy

- Splitting tasks into even-ish packages is necessary functionality
- This problem is np-complete

Approaches we looked at:

1. Round Robin: Assume that every Cell-Interaction has the same weight and distribute them via Round Robin
2. Greedy distribution: give the next job to the package that has the least work so far
3. Round Robin Threshold: Give Cell Interactions into one package until the number of interactions surpasses a threshold value (e.g. 10^4 interactions); continue in Round Robin fashion

Approach 1- 1D task model

- Store all tasks in one giant „task pool“ (e.g. a vector)
- Schedule freely
- Requires reduction
- Just one fork and join needed

6	7	8
3	4	5
0	1	2

```
tasks = {(0,1), (1,2), ... ,  
         (0,3), (3,6), ... ,  
         (0,4), (1,5), ... ,  
         (3,1), (4,2), ... }
```

Approach 2- Thread oriented 2D task model

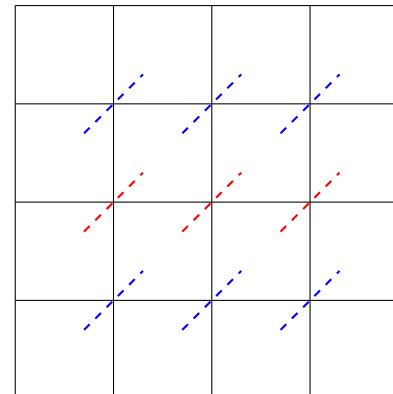
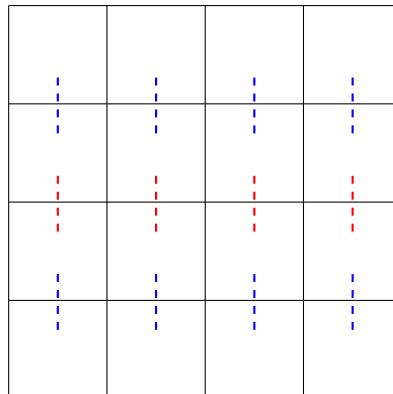
- Split up giant „task pool“ into num_threads jobs
- Use distribution strategy and information unavailable to scheduler (computing cost of each cell interaction) to distribute workload evenly
- Give one job to each thread
- Requires reduction
- Just one fork and join needed

6	7	8
3	4	5
0	1	2

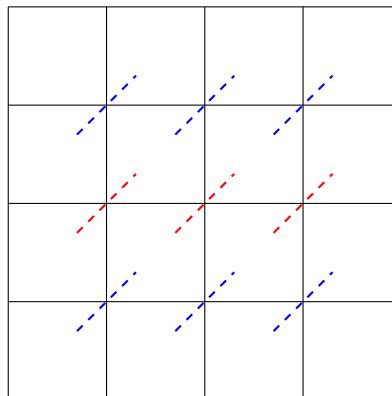
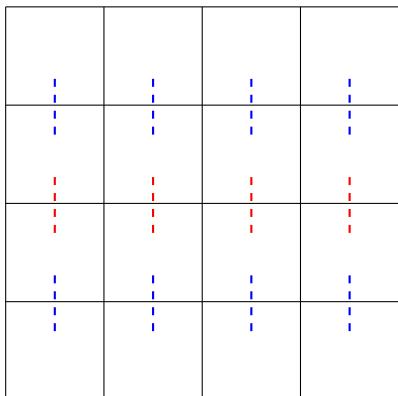
```
tasks = {{(0,1), (0,3), (3,6) ...},  
         {(1,2), (0,4), ...}}  
        ...  
    }
```

Approach 3- Color oriented 2D task model

- You need 13 „lines“ in the Cell-Algorithm to cover all neighbouring cell-interactions
- Idea: split up every line into 2 sets of edges to get 26 sets of edges that can be fully parallelized
- Fork and join 26 times
- No reduction required; no race condition in any of the 26 iterations possible
- Forks and joins needed



Approach 3- Color oriented 2D task model



```
tasks = {{vertical blue tasks},  
        {vertial red tasks},  
        {diagonal blue tasks},  
        {diagonal red tasks},  
        {other diagonal blue tasks},  
        ...  
};
```

Approach 4- 3D task model

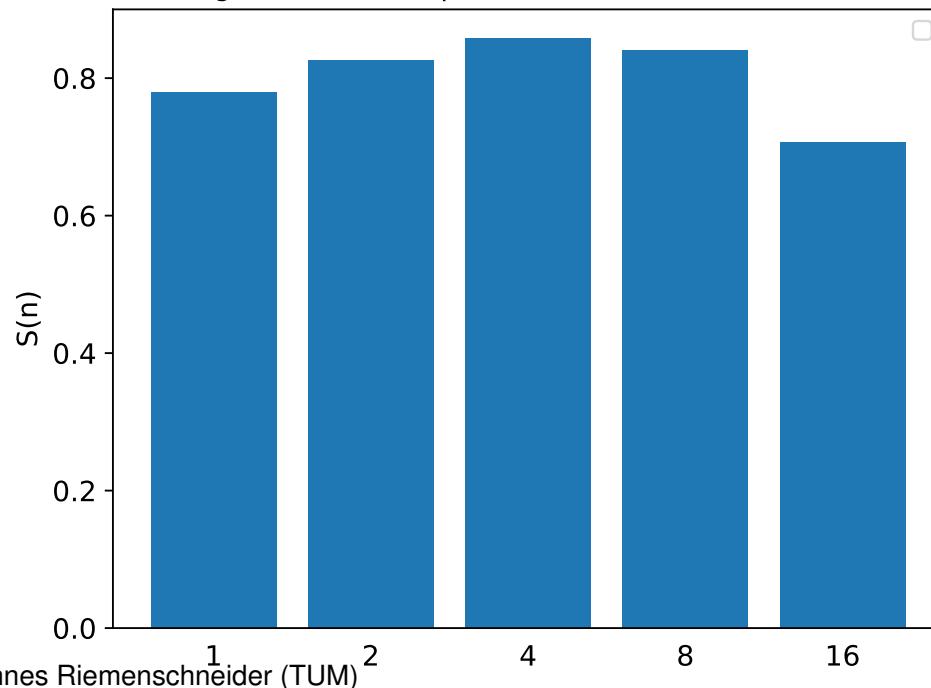
- Combination of thread oriented and color oriented 2D approaches
- Split up tasks into 26 fully parallelizable blocks
- Split those blocks into num_threads jobs; use a distribution strategy to balance workload
- No reduction required
- Forks and joins needed

Approach Comparison

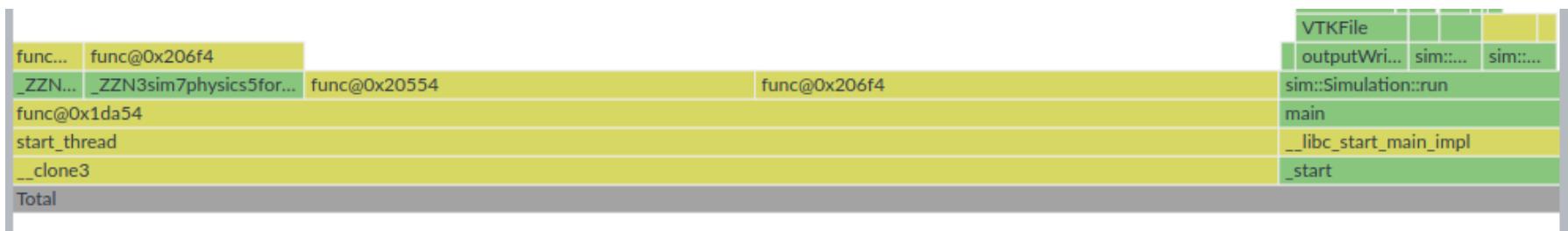
1D Tasks	2D thread tasks Greedy	2D colored tasks Round Robin	3D tasks Greedy	3D tasks Round Robin
Simplest approach	+ Potentially better scheduling than 1D tasks	+ Limit for parallelization corresponds to problem given theoretical limit	+ Potential to get the best out of both „2D worlds“	
+ Just one fork and join needed	+ Just one fork and join needed	- 26 forks and joins needed	- 26 forks and joins needed	
- Reduction needed	- Reduction needed	+ No reduction needed	+ No reduction needed	

Speedup of the 3D task model

SpeedUp of the 3D-Task approach with greedy distribution
and no padding between cells
Single-Threaded implementation used as reference



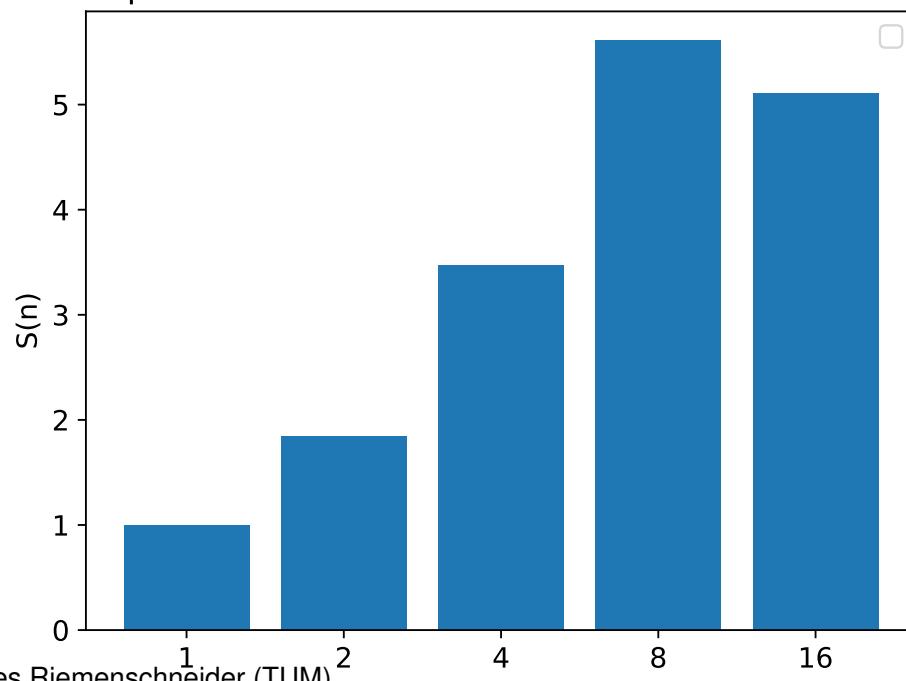
What happened?



- As you can see we only use a fraction of our time actually computing forces
- The majority of time is spent with OMP overhead
- Our conclusion: dividing the workload by $num_threads \cdot 26$ doesn't leave enough tasks in each job

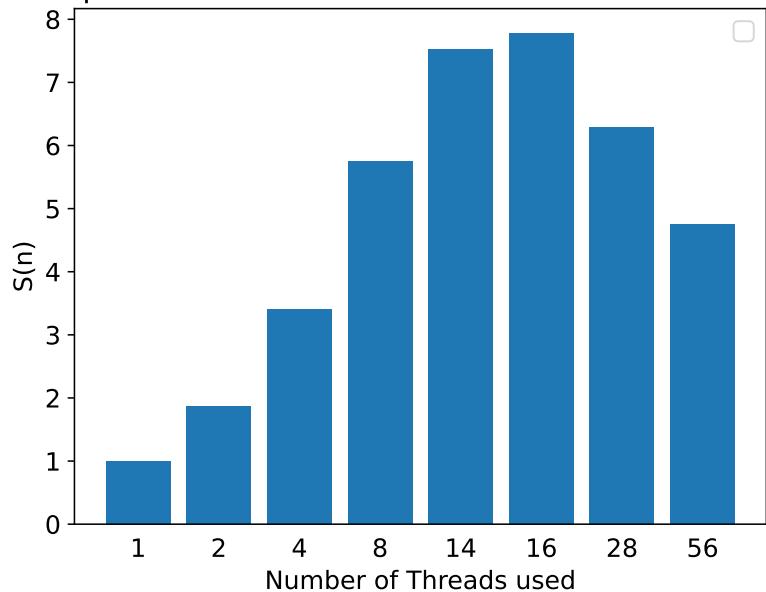
Speedup of the 1D approach

SpeedUp of the 1D-Task approach with no padding between cell
OMP-implementation executed with one Thread is reference

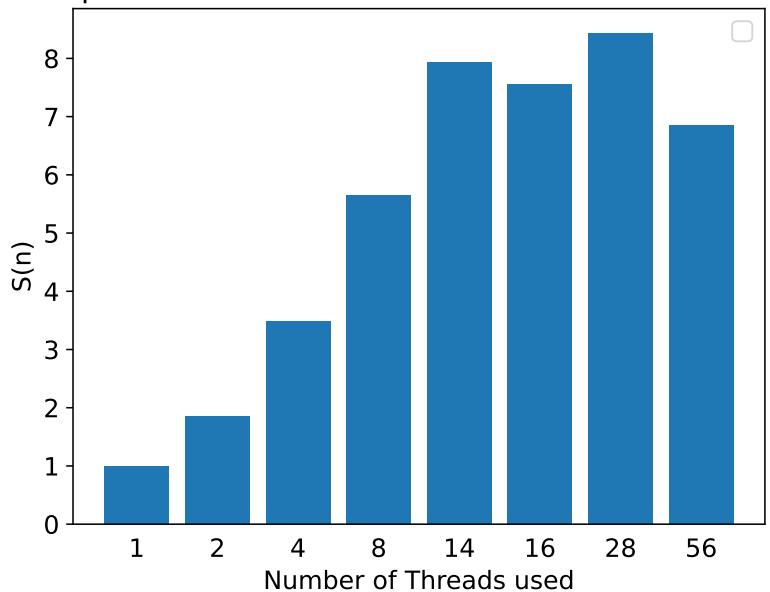


Speedups of the 2D approaches

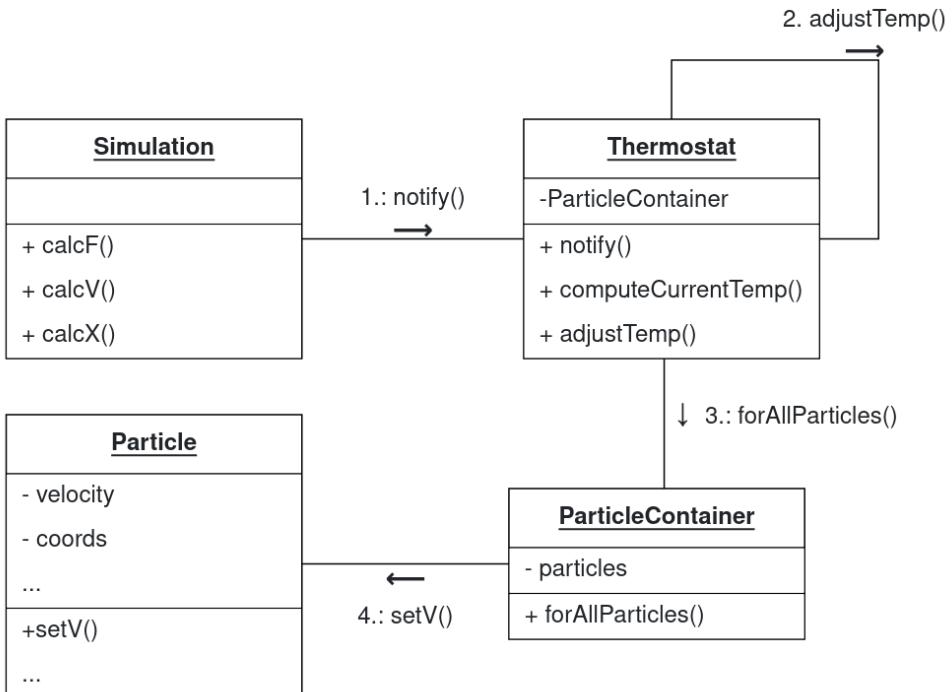
SpeedUp of the 2D task model with greedy distribution
OMP-implementation executed with one Thread is reference



SpeedUp of the 2D color oriented approach
OMP-implementation executed with one Thread is reference



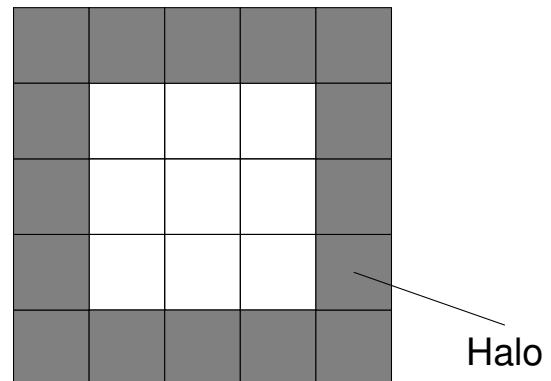
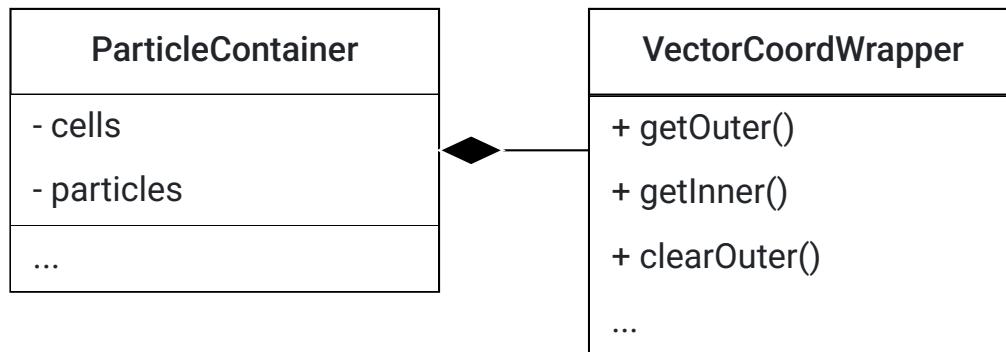
Thermostat



Adapting ParticleContainer for periodic bounds

Idea:

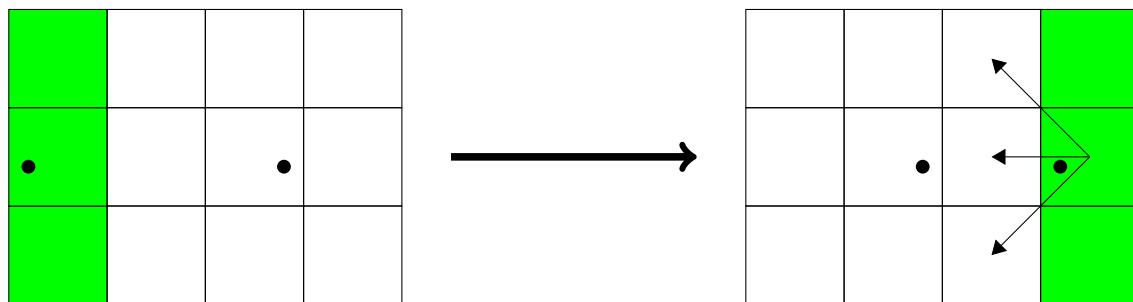
- Provide virtual cells around the actual domain for anyone who needs it
- Existence of additional cells is invisible with old interface



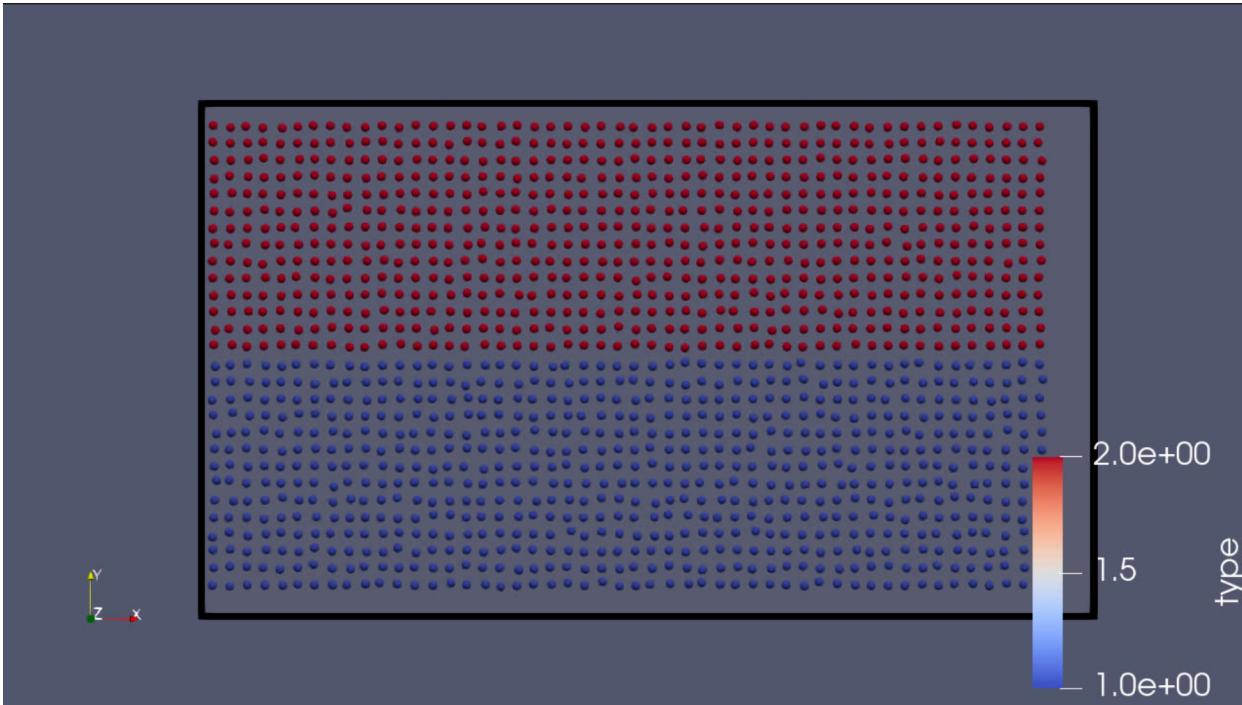
Boundary conditions

Idea:

1. Temporarily move all particles next to Boundary of the other side
2. Let Neighbouring cells interact

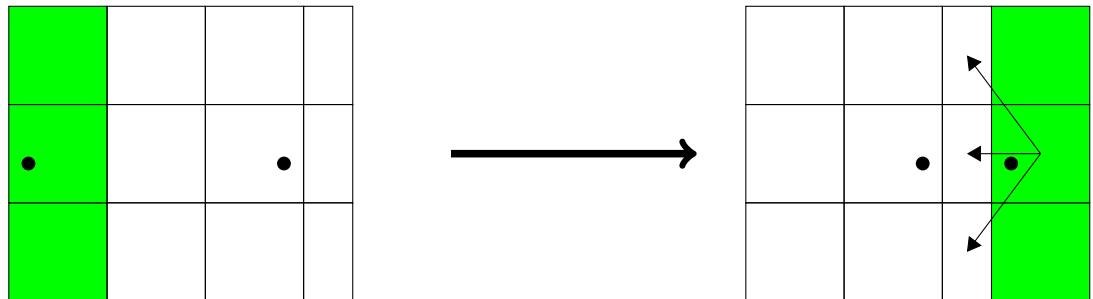


The result



The problem

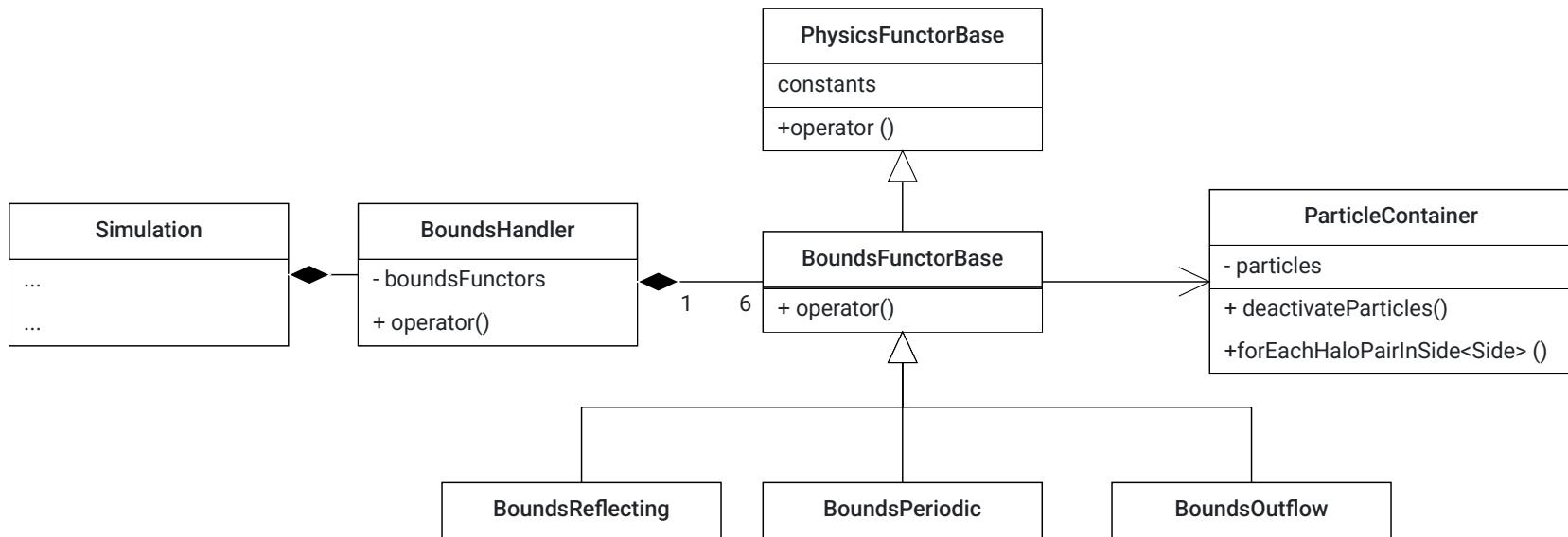
1. Outer boxes may not have the expected sidelengths
2. Interacting with neighbouring cells \Rightarrow Catching everything in r_{cutoff}



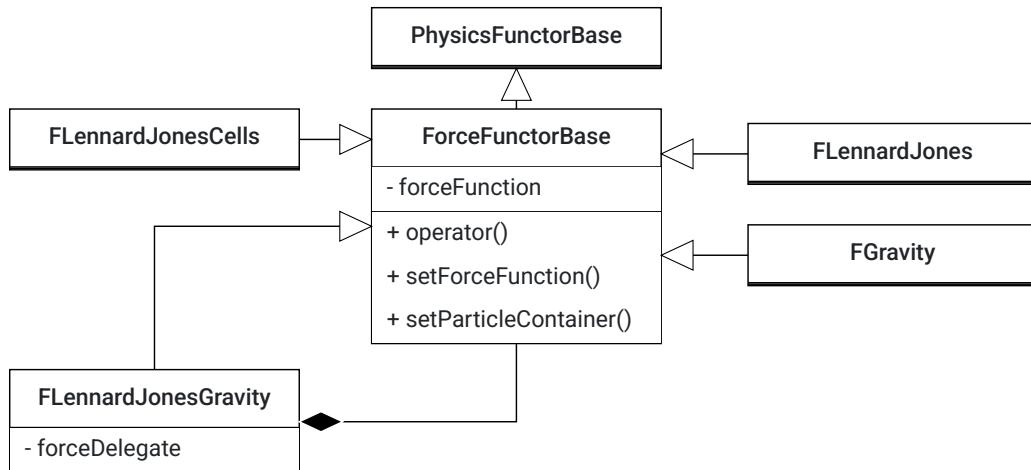
\Rightarrow Interact with one more „Cellblock“ in that direction

Adding Periodic bounds

This slide should look very familiar to Assignment 3



Adding Gravitational Force



```
FLennardJonesGravity :: operator ()(){
    forceDelegate->operator ();
    particleContainer.forAllParticles ([]( auto& p){
        p.force[1] += p.m * gGrav;
    });
}
```

Optimizations 1

As mentioned in Assignment 3 our ParticleContainer does not contain Particle-structs anymore. Keeping the old interface lead to the following method:

```
void ParticleContainer::forAllParticles(void(*function)(Particle &)) {  
    for (unsigned long index: activeParticles) {  
        Particle p;  
        loadParticle(p, index);  
        function(p);  
        storeParticle(p, index);  
    }  
}
```

⇒ rewriting old code where this method got used was a major improvement

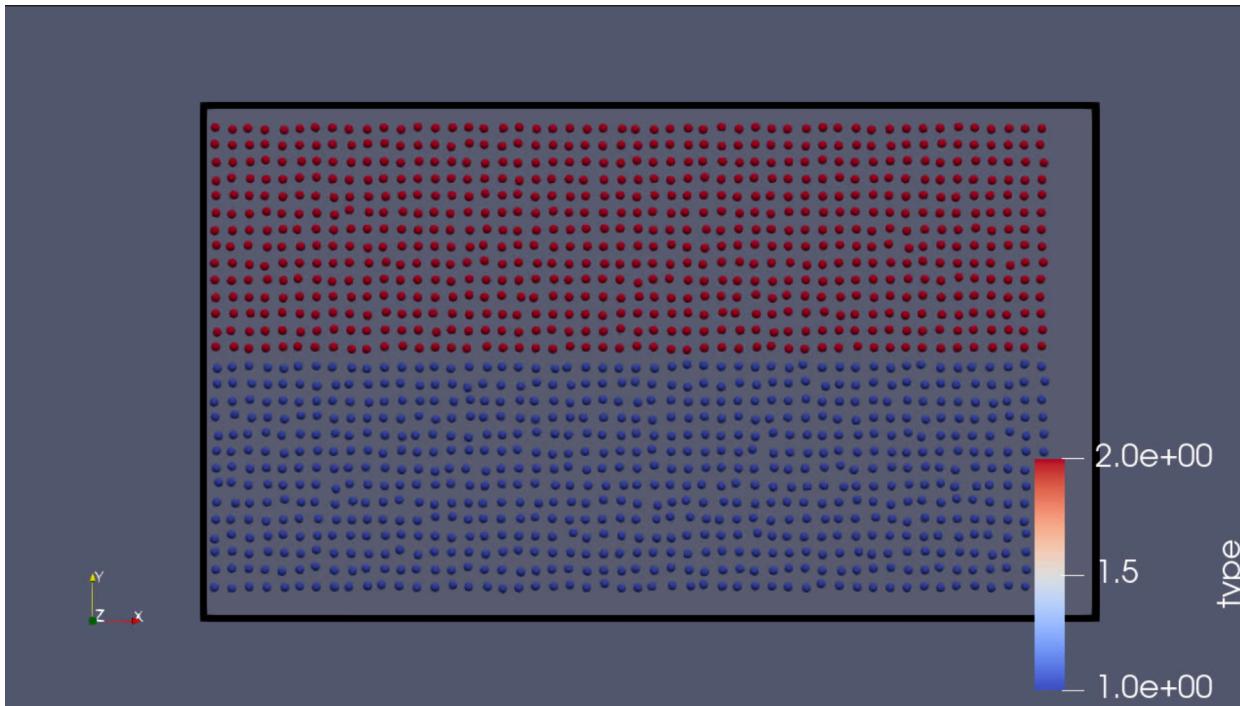
Optimizations 2

sim::Simulation::runBenchmark	100.0%	0s	MolSim	sim::Simulati...	0x63282
▼ sim::physics::force::FLennardJonesCells::operator()	88.2%	0s	MolSim	sim::physics::...	0x143b94
▼ ParticleContainer::forAllDistinctCellNeighbours<sim::physics::force::FL	81.6%	0.024s	MolSim	ParticleCont...	0x143dd6
► sim::physics::force::FLennardJonesCells::operator()(void)::(lambda(s	80.9%	0.640s	MolSim	sim::physics::...	0x143a00
► ParticleContainer::VectorCoordWrapper::operator[]	0.4%	0.012s	MolSim	ParticleCont...	0x76be6

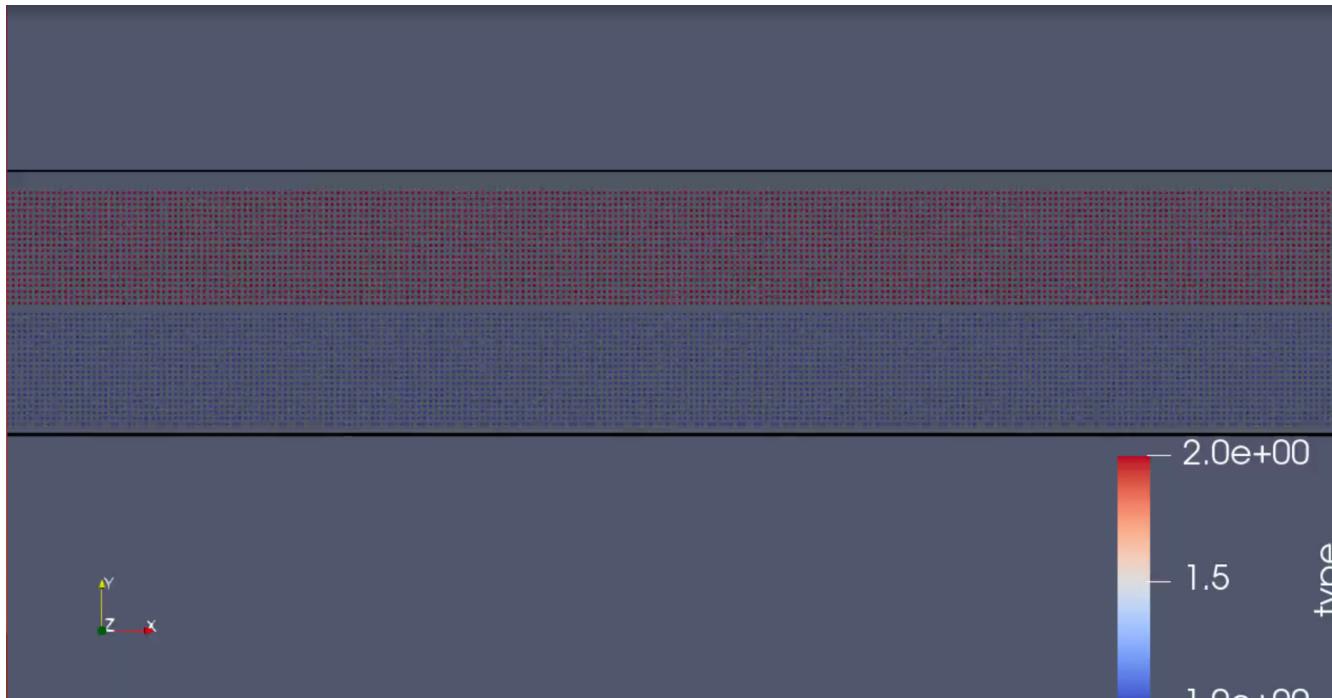
- Force calculation takes a significant portion of CPU time
- Force between two particles in Force-Functors got represented as lambda expression

⇒ Represent force as static function instead

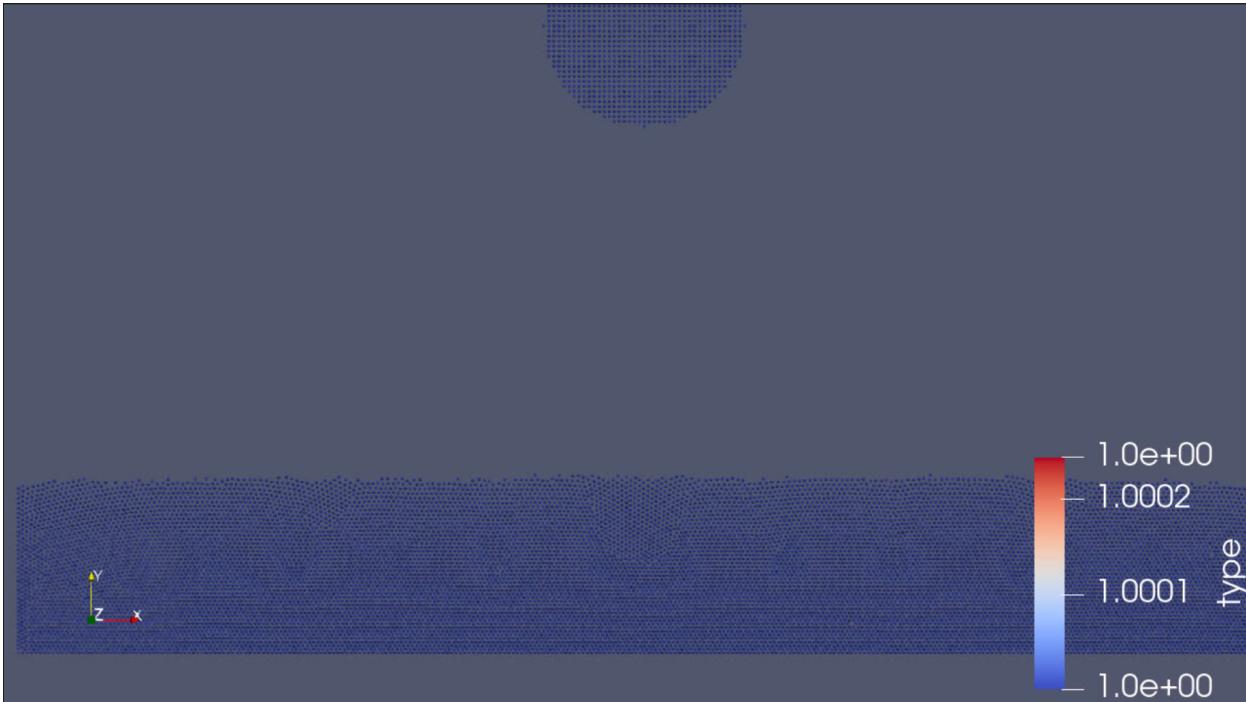
Small Rayleigh-Taylor instability



Rayleigh-Taylor instability



Falling drop



Serial Benchmarks

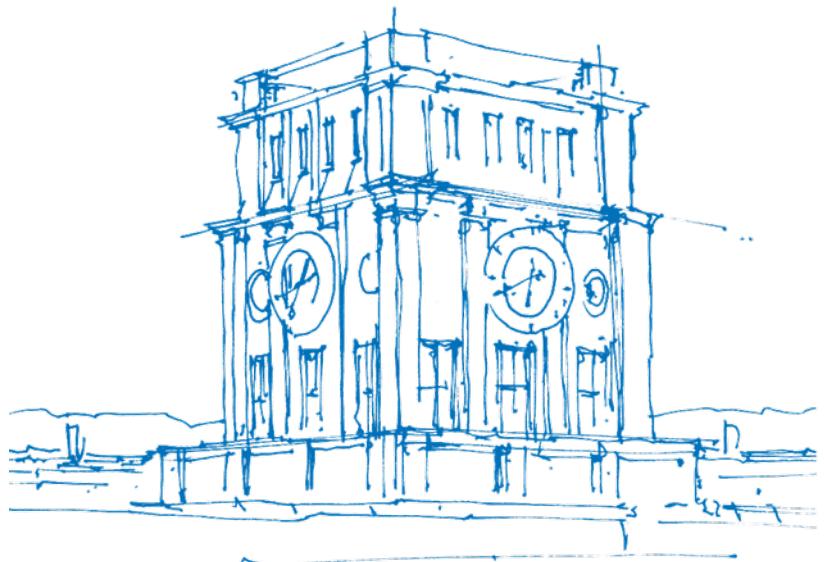
- As already mentioned building with the Intel compiler didn't work even with major time investments
- The option „-slow“ lets the code run in the pre-optimized state

Options	MMU/s Cluster	MMU/s Local
-slow -O2	0.0087	$\geq 1/6 \cdot 0.036 \wedge \leq 1/2 \cdot 0.036$
-O2	0.0087	0.036
-O0	0.0087	0.036

Since these measurements are so close together and significantly smaller than our local measurements, we assume that something went wrong on the cluster.

Roadblocks

- Compiling and running jobs on the cluster turned out to be a nightmare
- Intel compiler broke us trying to unbreak him
- Searching for bugs that may or may not be there (bouncy particles in Rayleigh-Taylor)
- Searching for bugs that definitely are there (see Boundary conditions)
- Large time investments in order to get tools to run



TUM Uhrenturm

Recreating Profiling

1. `mkdir build`
2. `cmake ..`
3. `make ProfileMolSim` or `make CXX_FLAGS+=Dslow -std=c++20"ProfileMolSim`
4. `./ProfileMolSim ../input/[file_you_want_to_profile]`
5. `gprof ProfileMolSim gmon.out > profile-data.txt`