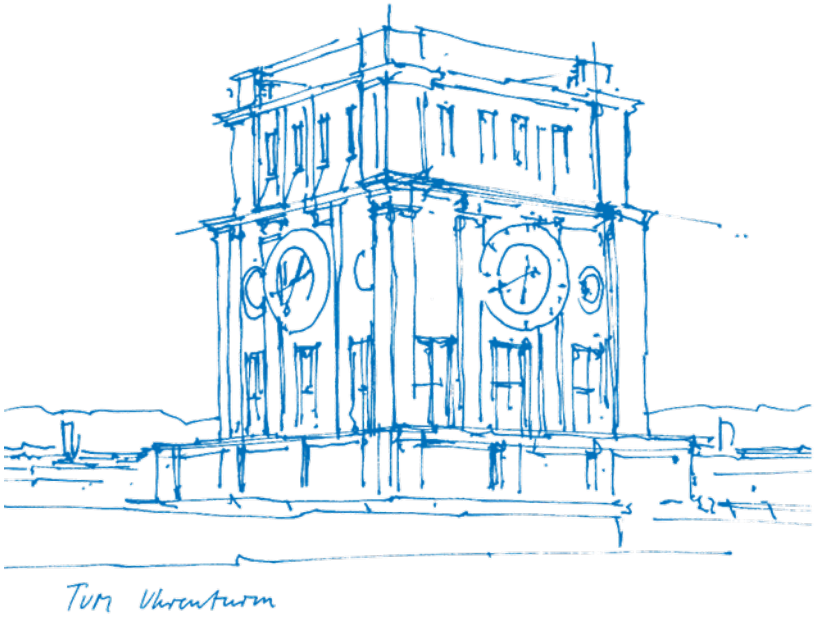
Molecular Dynamics - Assignment 5

Alex Hocks Jan Hampe Johannes Riemenschneider
Technische Universität München
TUM CIT
Lehrstuhl für wissenschaftliches Rechnen
2. Februar 2023



Implementation of Membranes

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

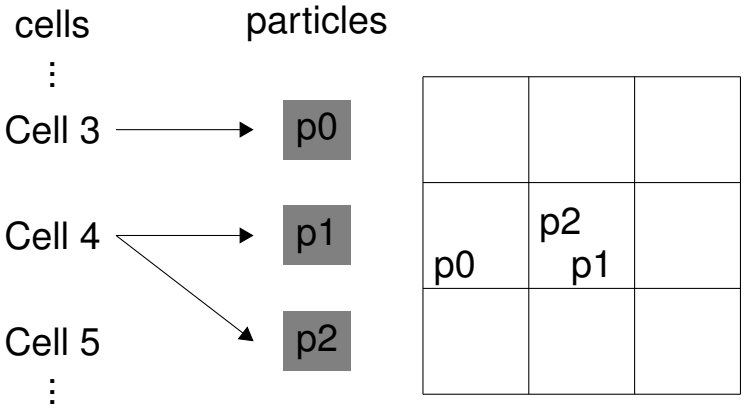
Implementation of immovable Particles

- 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 Assigment 3):

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

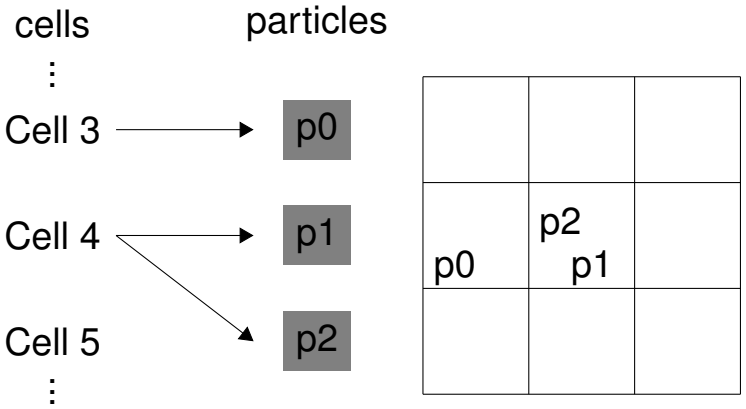


Our changes to avoid false sharing:

Preparation for Multithreading- Refactoring the Particle Container

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

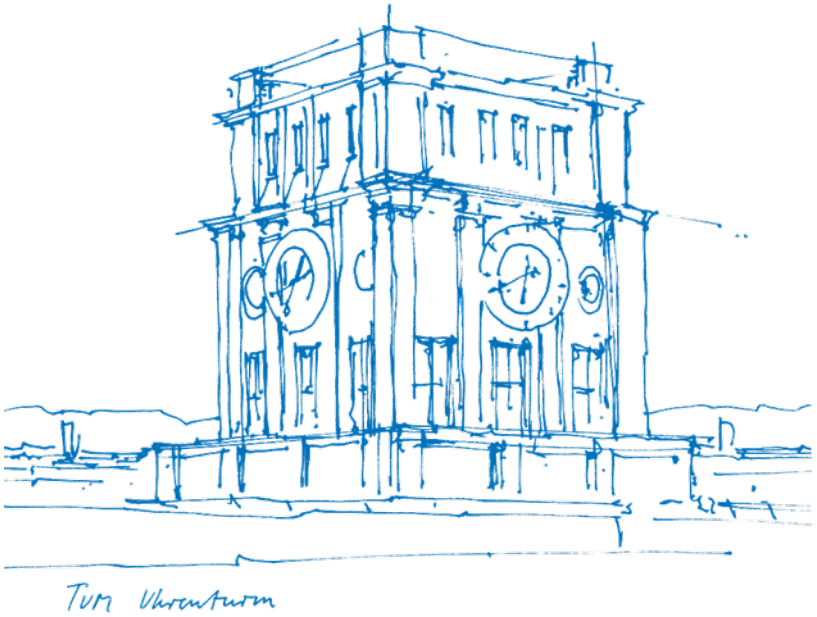
- Particles get stored in one giant vector
- Each Cell keeps references to their particles
- No sorting or copying takes place



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 Oddyssey



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.

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.

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:

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

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

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

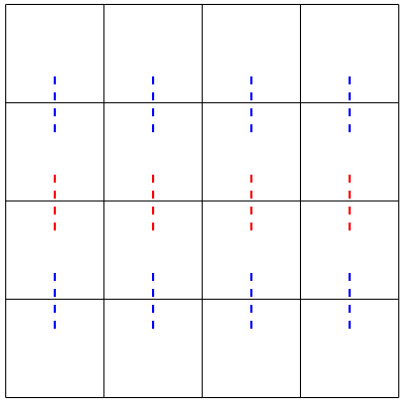
- Split up giant „task pool“ into num_threads jobs
- Use distribution strategy and information unavailable to scheduler (computing cost of each cell interaction) to distribute workload evenly
- Give one job to each thread
- Requires reduction
- Just one fork and join needed

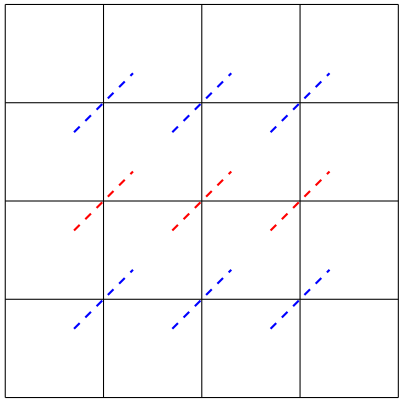
6	7	8
3	4	5
0	1	2

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

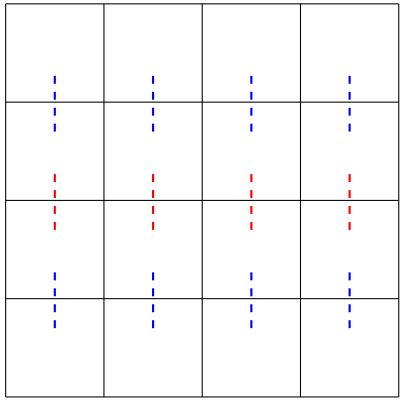
Approach 3- Color oriented 2D task model

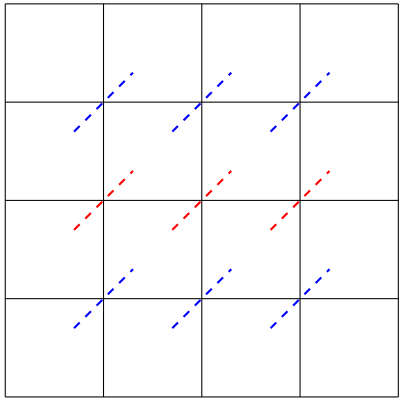
- You need 13 „lines“ in the Cell-Algorithm to cover all neighbouring cell-interactions
- Idea: split up every line into 2 sets of edges to get 26 sets of edges that can be fully parallelized
- Fork and join 26 times
- No reduction required; no race condition in any of the 26 iterations possible





Approach 3- Color oriented 2D task model





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

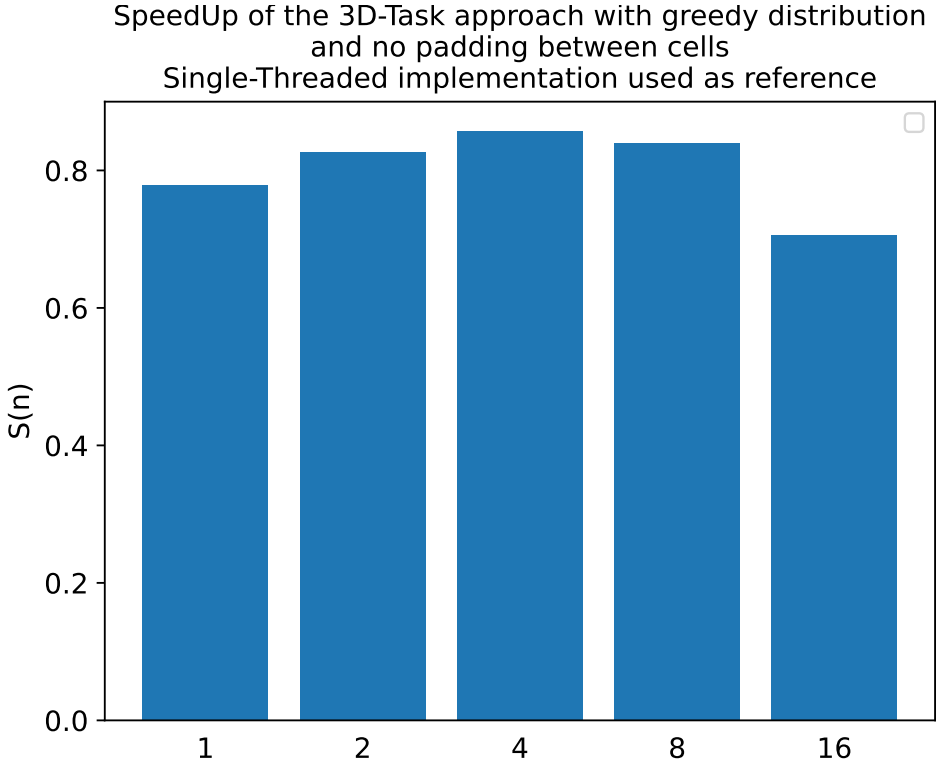
Approach 4- 3D task model

- 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
- 26 forks and joins needed

Approach Comparison

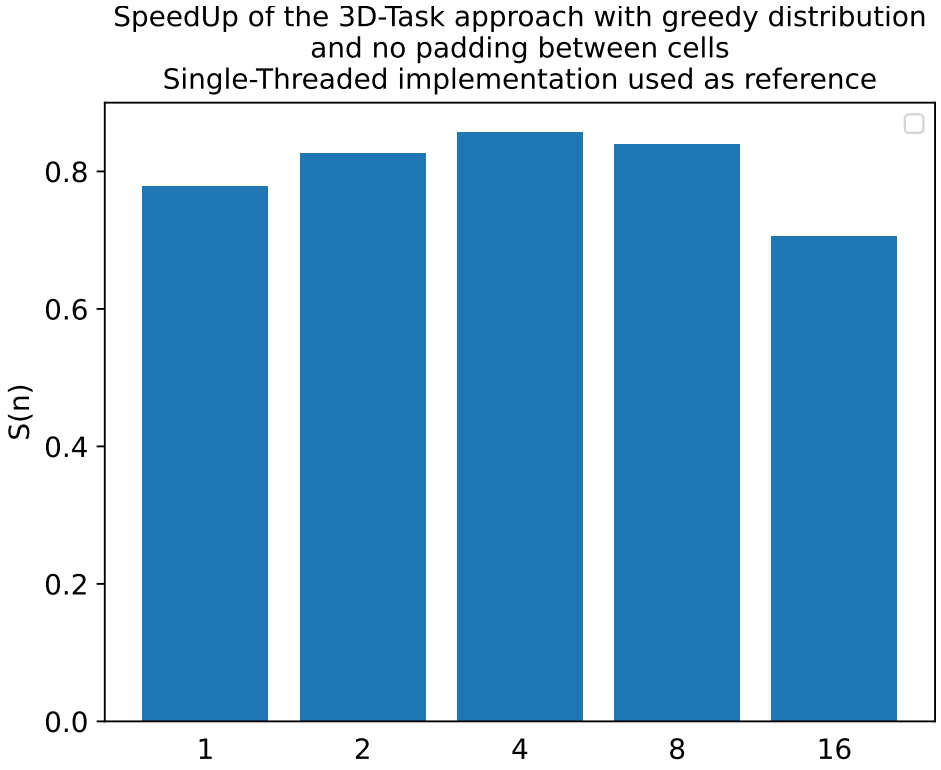
1D Tasks	2D thread tasks		2D colored tasks	3D tasks	
	Greedy	Round Robin		Greedy	Round Robin
Simplest approach	+ Potentially better scheduling than 1D tasks		+ Limit for parallelization corresponds to theoretical limit given by the problem	+ Potential to get the best out of both „2D worlds“	
+ Just one 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 (local build)

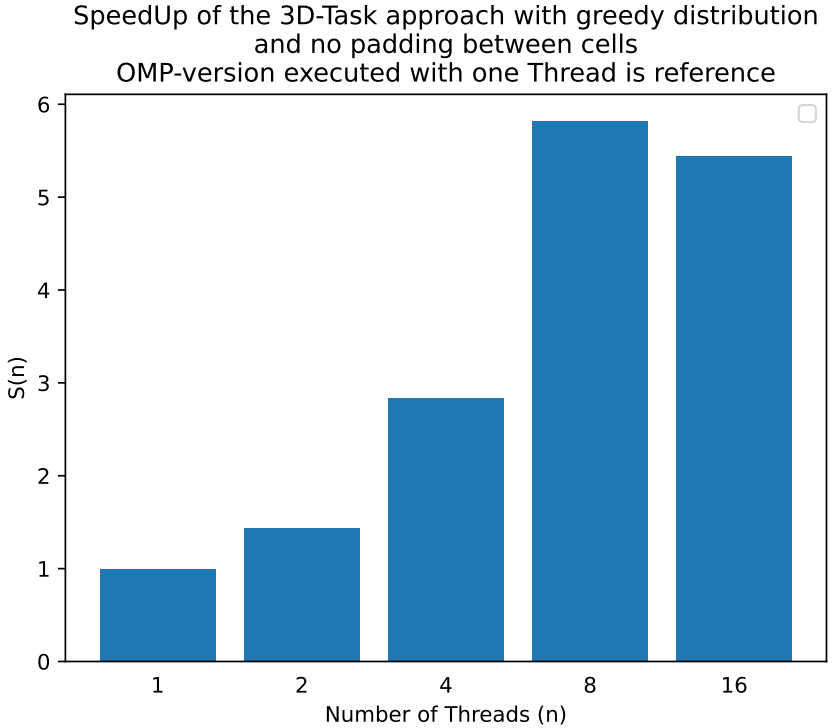


Speedup of the 3D task model (local build)

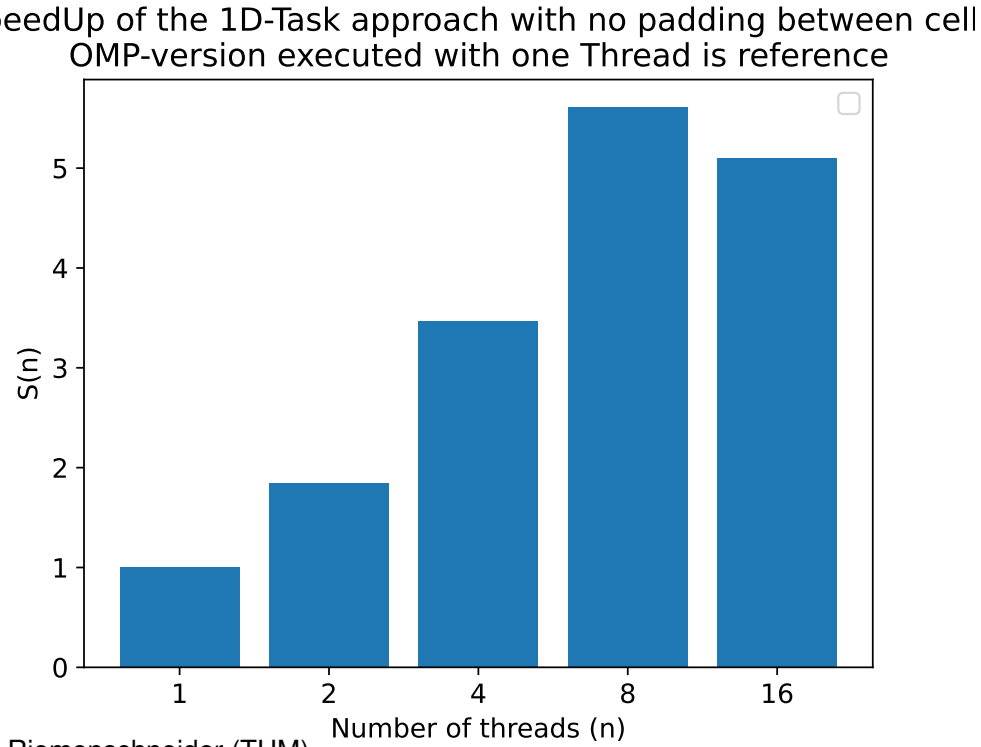
That's the speedup with a broken task model that gives almost all the workload to one thread



Actual speedup of the 3D approach (local build)

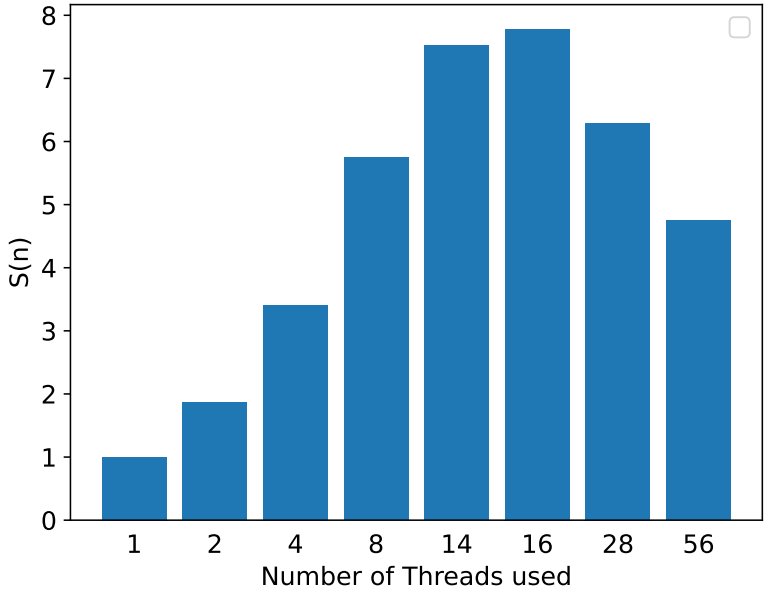


Speedup of the 1D approach (local build)

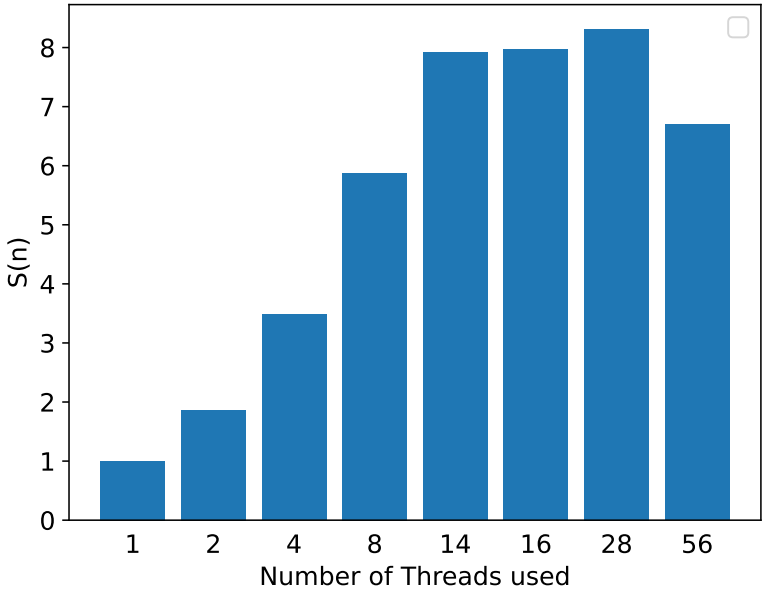


Speedups of the 2D approaches (build on cluster)

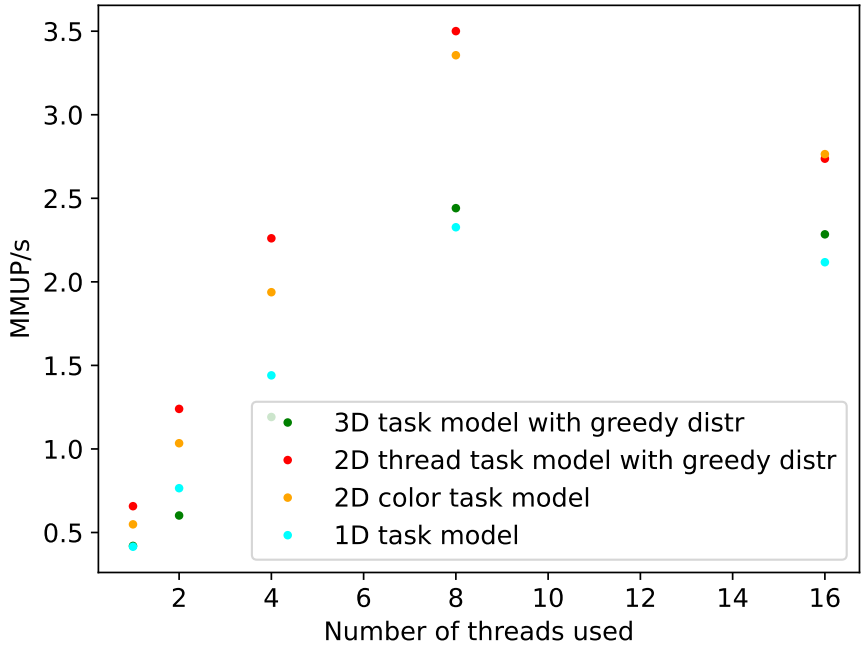
SpeedUp of the 2D task model with greedy distribution
OMP version executed with one Thread is reference



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

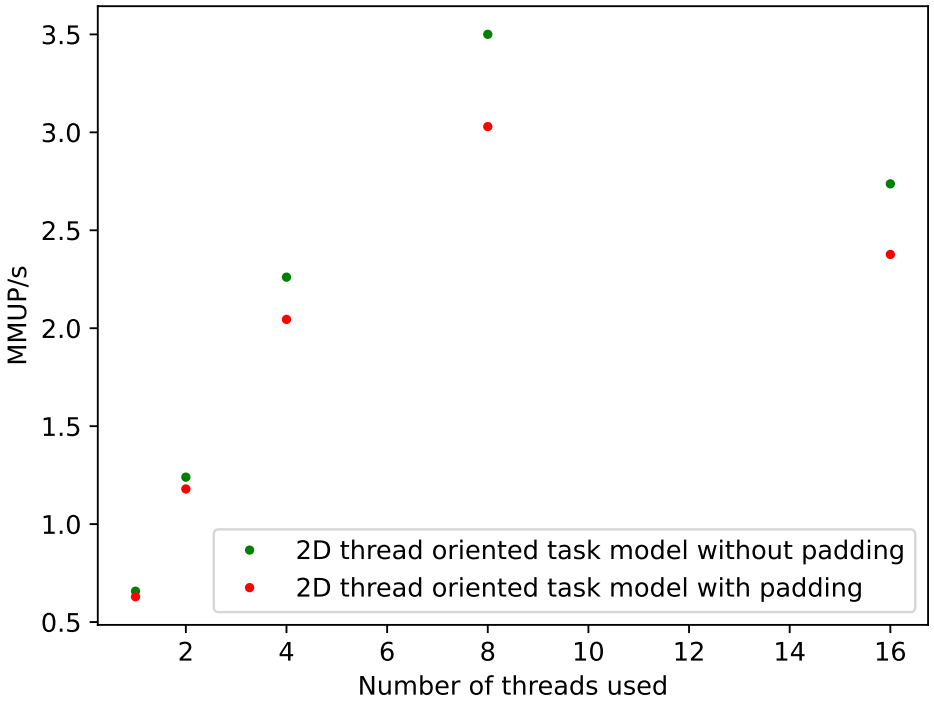


All approaches in comparison (local build)



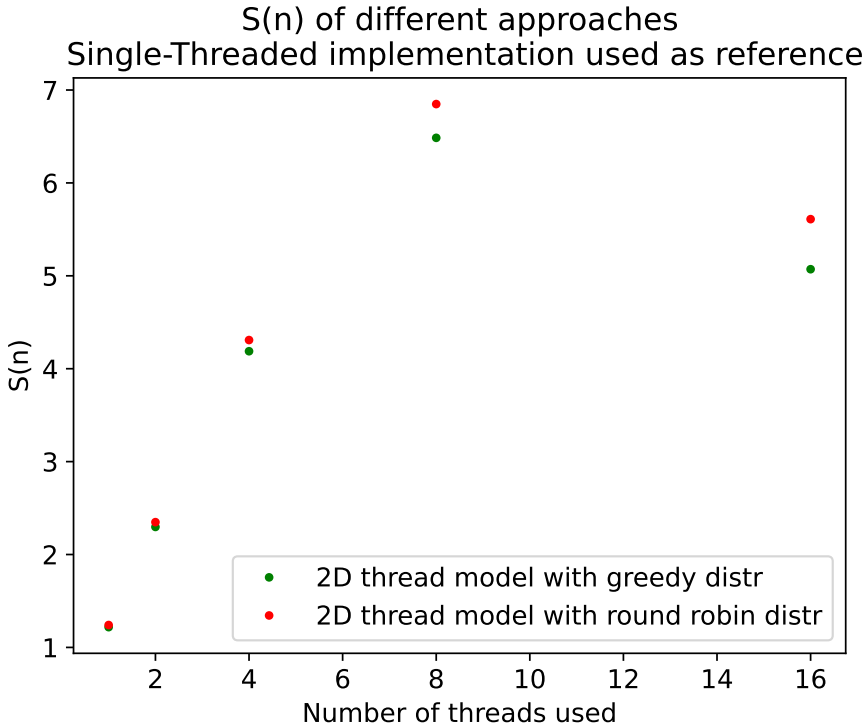
The price of padding

- As you can see padding adds a measurable overhead
- Even in cases where we expected false sharing to occur disabling padding gave a performance boost
- $\Longrightarrow$ no padding became the default option



Comparing distribution strategies

- With a good threshold value round robin has a better performance
- We still chose greedy distribution as default option since it doesn't need that type of „ prerun-tuning“



Analyzing the speedup pleateau

Observations:

- Behaviour occurs with and without padding

Analyzing the speedup pleateau

Observations:

- Behaviour occurs with and without padding
- Number of threads needed to reach speedup maximum varies from computer to computer

Analyzing the speedup pleateau

Observations:

- Behaviour occurs with and without padding
- Number of threads needed to reach speedup maximum varies from computer to computer
- Changing the workload per iteration step also changes the speedup maximum

Analyzing the speedup pleateau

Observations:

- Behaviour occurs with and without padding
- Number of threads needed to reach speedup maximum varies from computer to computer
- Changing the workload per iteration step also changes the speedup maximum
- The more threads you add the more runtime gets used for OMP overhead methods

Analyzing the speedup pleateau

Observations:

- Behaviour occurs with and without padding
- Number of threads needed to reach speedup maximum varies from computer to computer
- Changing the workload per iteration step also changes the speedup maximum
- The more threads you add the more runtime gets used for OMP overhead methods

Conclusion:
Not checking whether an additional thread has enough workload to be created $\Longrightarrow$ creating threads where the speedup gets consumed by the OMP overhead

Additional parallelizations

Parallelization of:

- **Bounds handling**: Slight performance increase
- **Velocity calculation**: Performance decrease (only num_particles operations per iteration)
- **Position calculation**: Performance decrease (only num_particles operations per iteration)

with no design decisions worth mentioning.

$\Longrightarrow$ no additional parallelization of other code areas with comparably small workloads

NanoFlow

Issues, bugs and other timesinks

- Our Computation of position

$$x_i(t_{n+1}) = x_i(t_n) + \Delta t \cdot v_i(t_n) + (\Delta t)^2 \cdot \frac{F_i(t_{n-1})}{2m_i} \tag{1}$$

turned out to be the bug with the worst characters changed / time spent ratio.

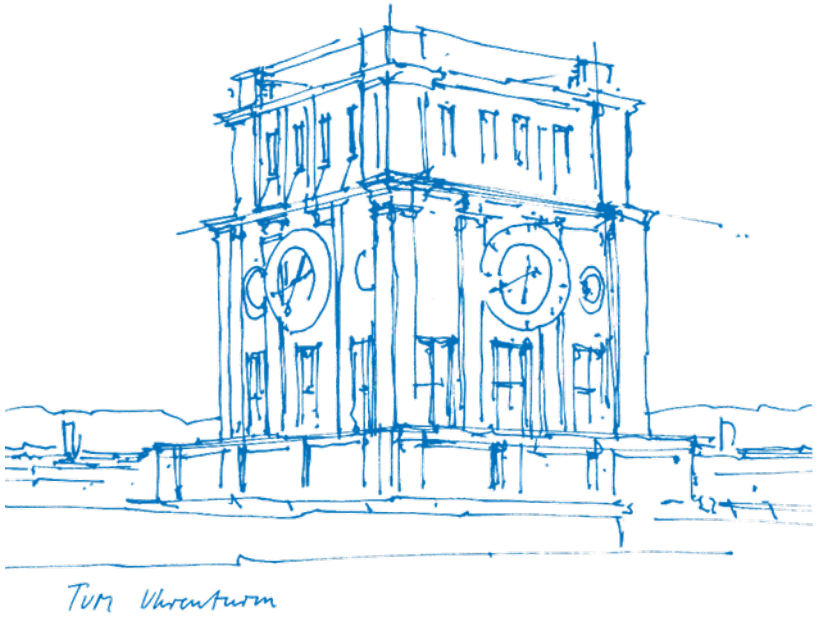
Issues, bugs and other timesinks

- Our Computation of position

$$x_i(t_{n+1}) = x_i(t_n) + \Delta t \cdot v_i(t_n) + (\Delta t)^2 \cdot \frac{F_i(t_{n-1})}{2m_i} \tag{1}$$

  turned out to be the bug with the worst characters changed / time spent ratio.

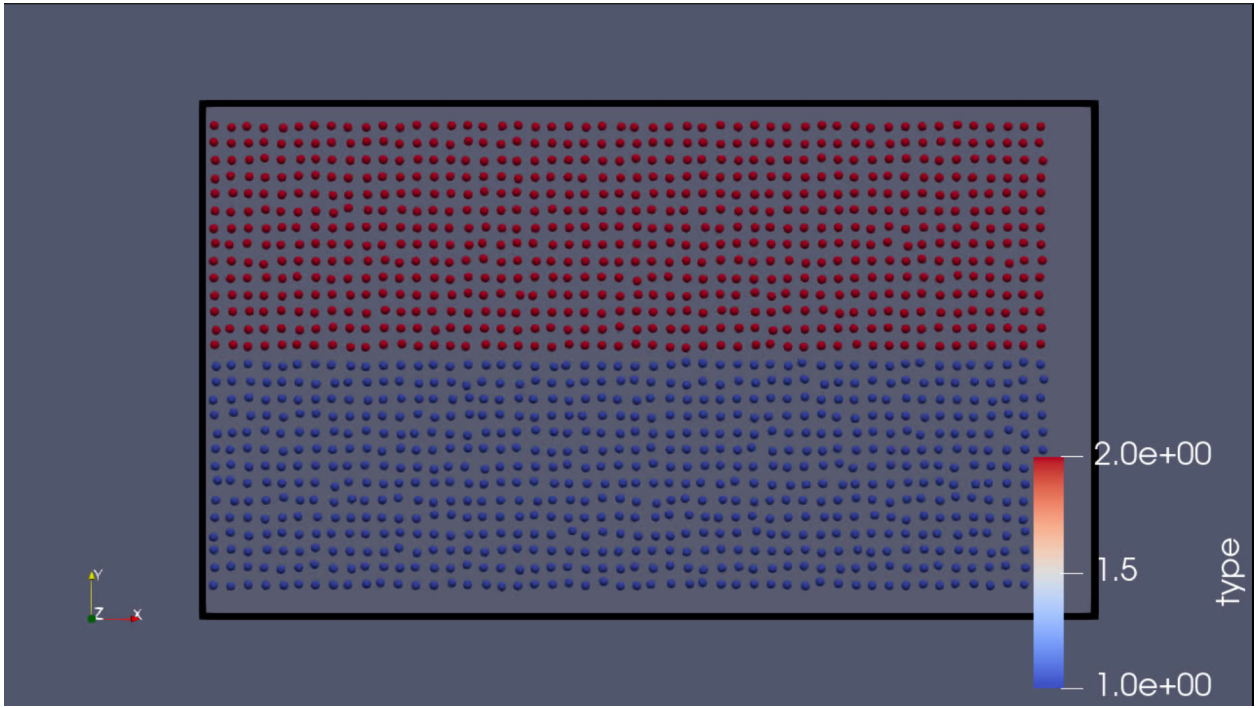
- Understanding the vtune-output of the 3D approach took a considerable amount of time. Looking back at it we don't really know why.


TUM Uhrenturm

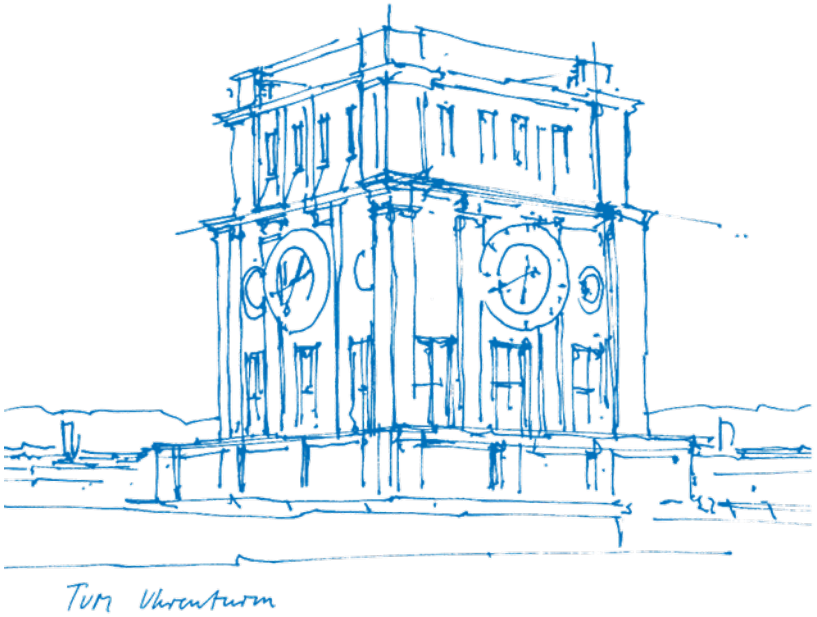
Choosing a parallelization implementation

1. `mkdir build`, `cd build`
2. Pick one: `cmake ..`, `cmake -Dround_robin_distr=1 ..`, `cmake -Dtask_oriented_2d=1 ..`, `cmake -Done_dim_tasks=1 ..`, `cmake -Dthree_dim_tasks=1 ..`, `cmake -Dthree_dim_tasks=1 -Dround_robin_distr=1 ..`
3. `make`
4. `./MolSim [path to input file]` for normal execution or `./MolSim [path to input file] -bench file -i [number of iterations]` to use benchmark mode

The result



Additional Graphs

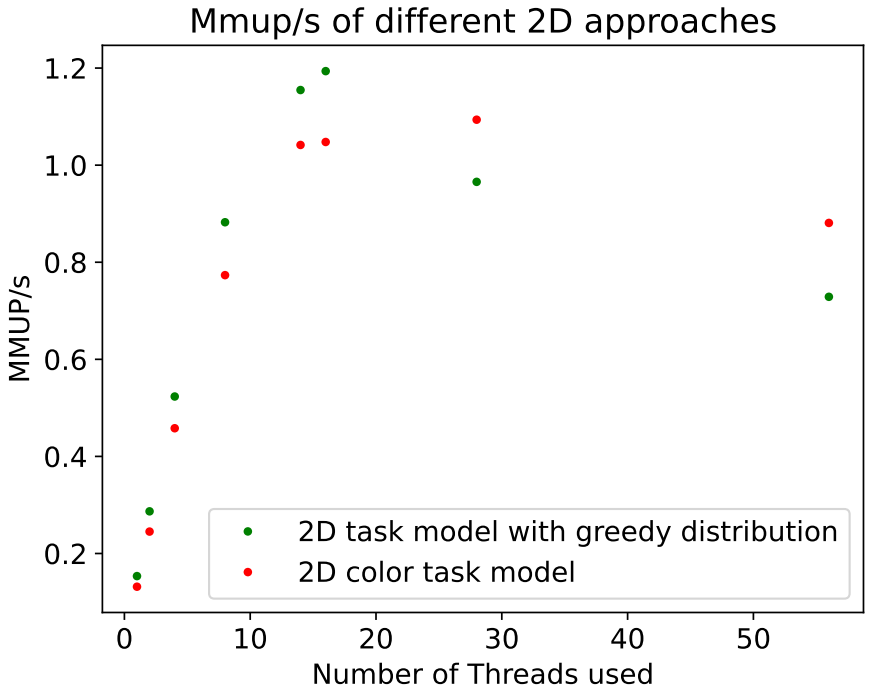


Local builds

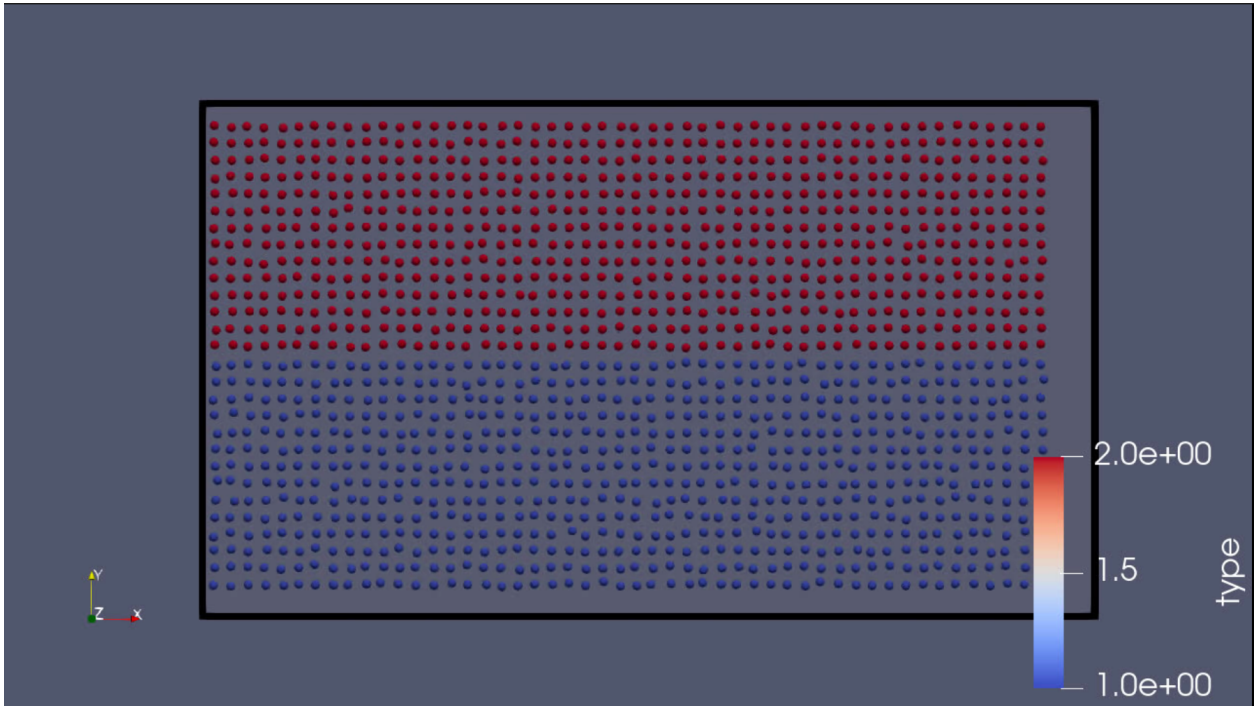
Since the cluster was very crowded we let some measurements run on a local computer.
Hardware details:
`i7 12700 KF @ 4,7 GHz, 64 GB RAM @ 3200 MT/s`

content...

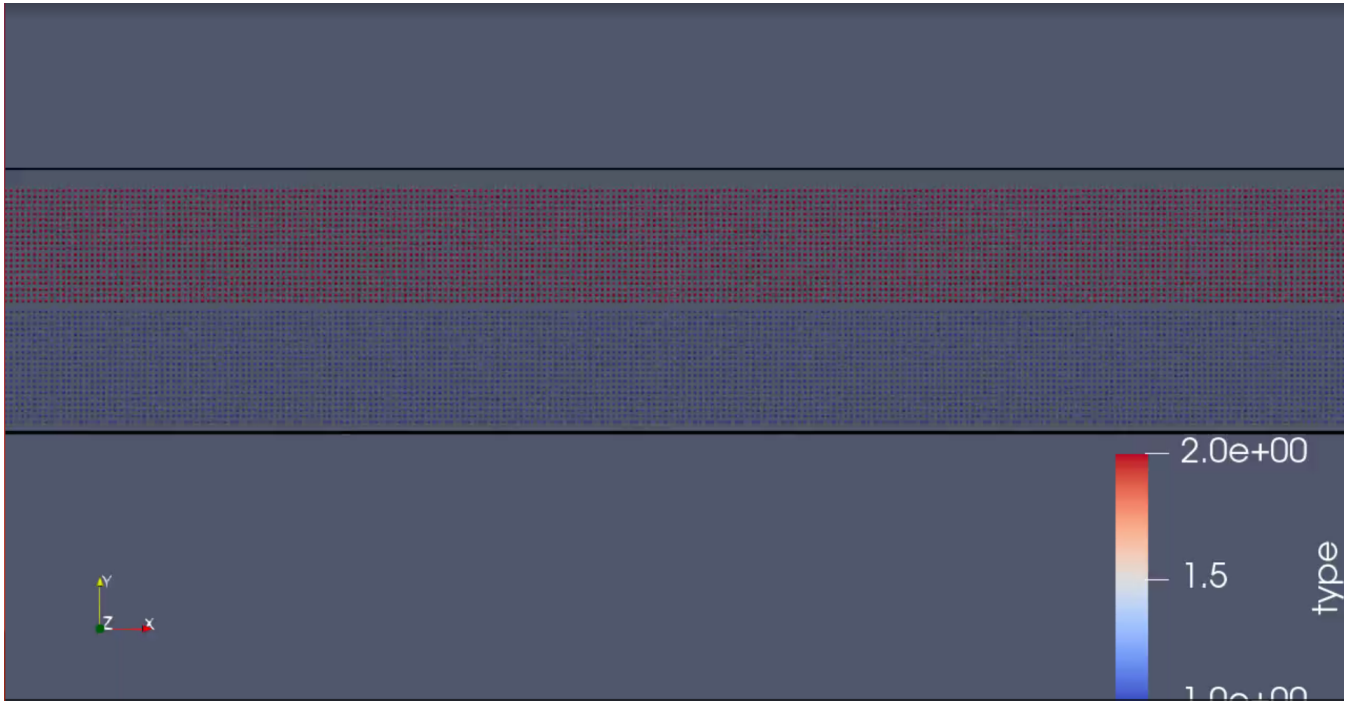
Builds on cluster



Small Rayleigh-Taylor instability



Rayleigh-Taylor instability



Falling drop