Parallelization of particle based systems

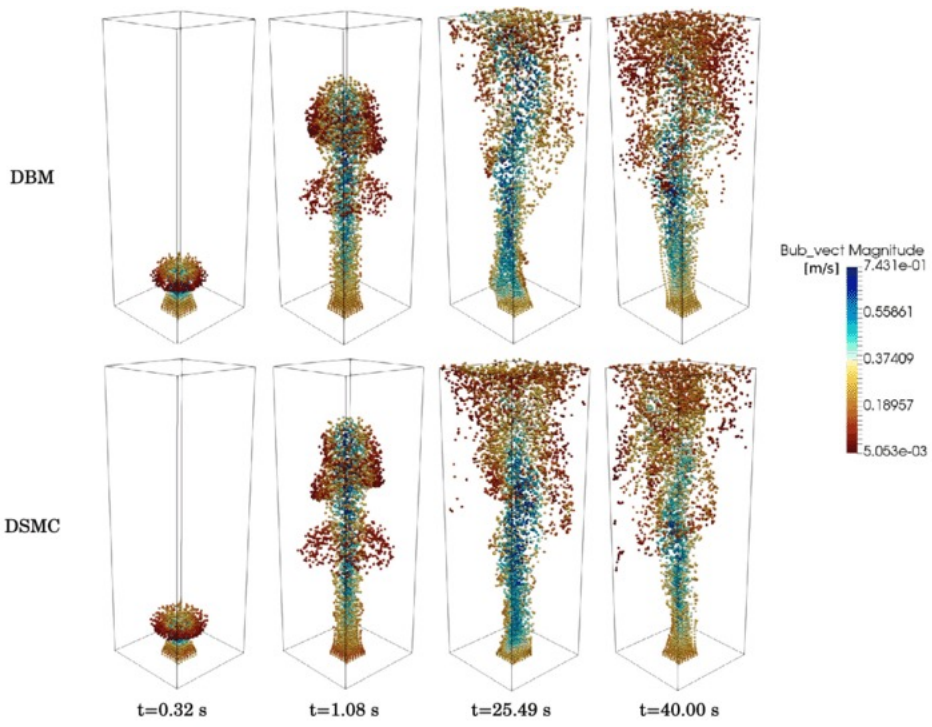
A distributed, shared and hybrid perspective

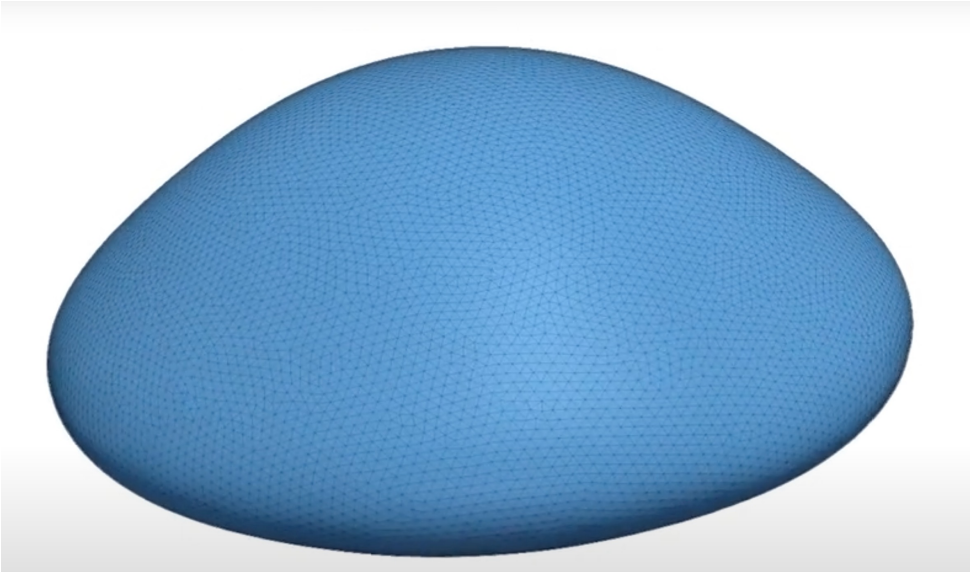
Contents

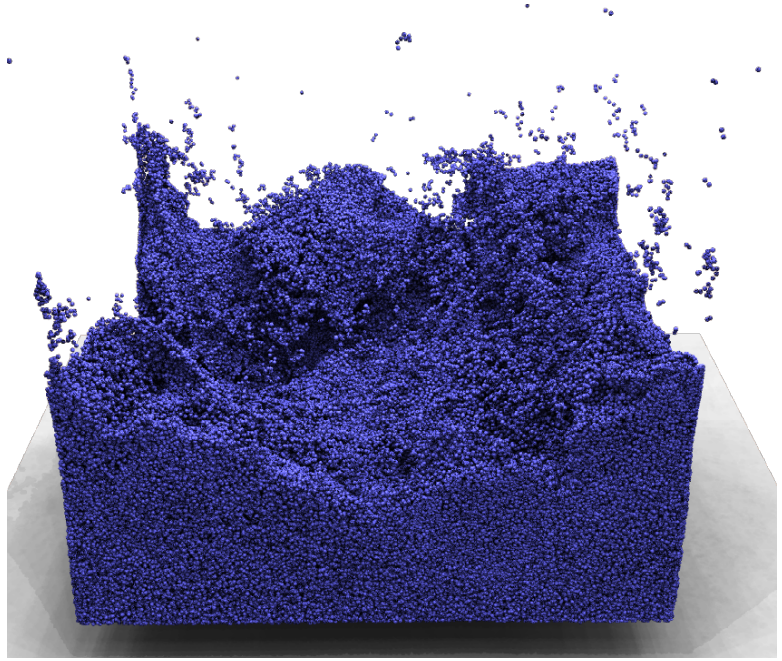
What are Lagrangian systems?

- Systems with moving entities in a static frame
- Interaction laws/rules
- External force fields
- May have an inlet or outlet or a system complete in itself
- These entities can be
  - Particles
  - Meshed elements
  - Coarse grained elements

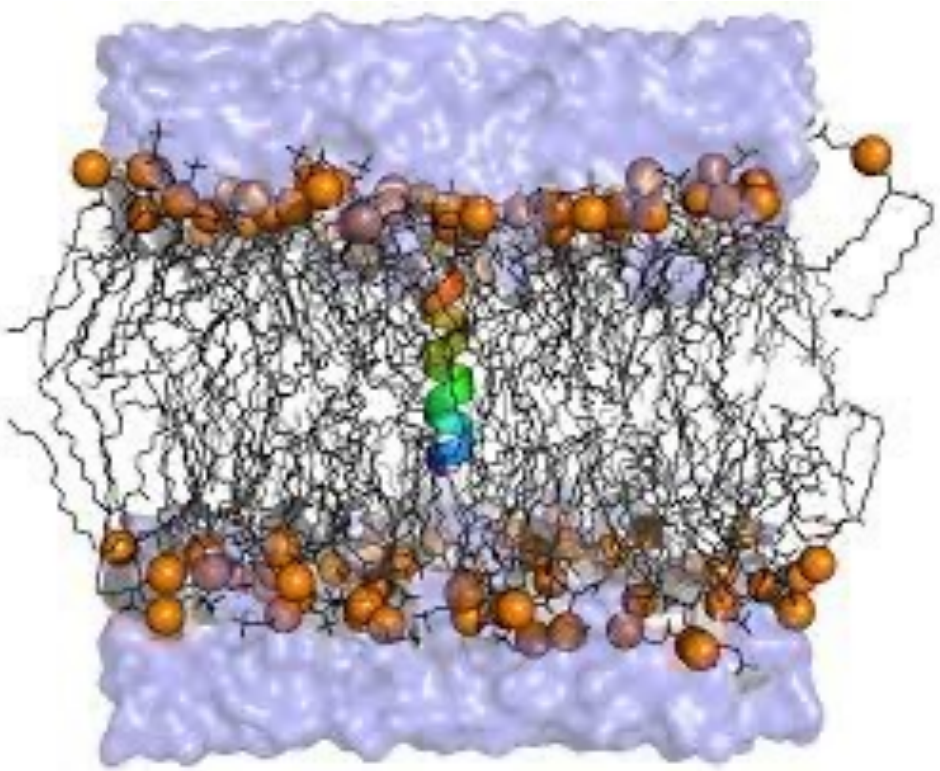
Typical examples



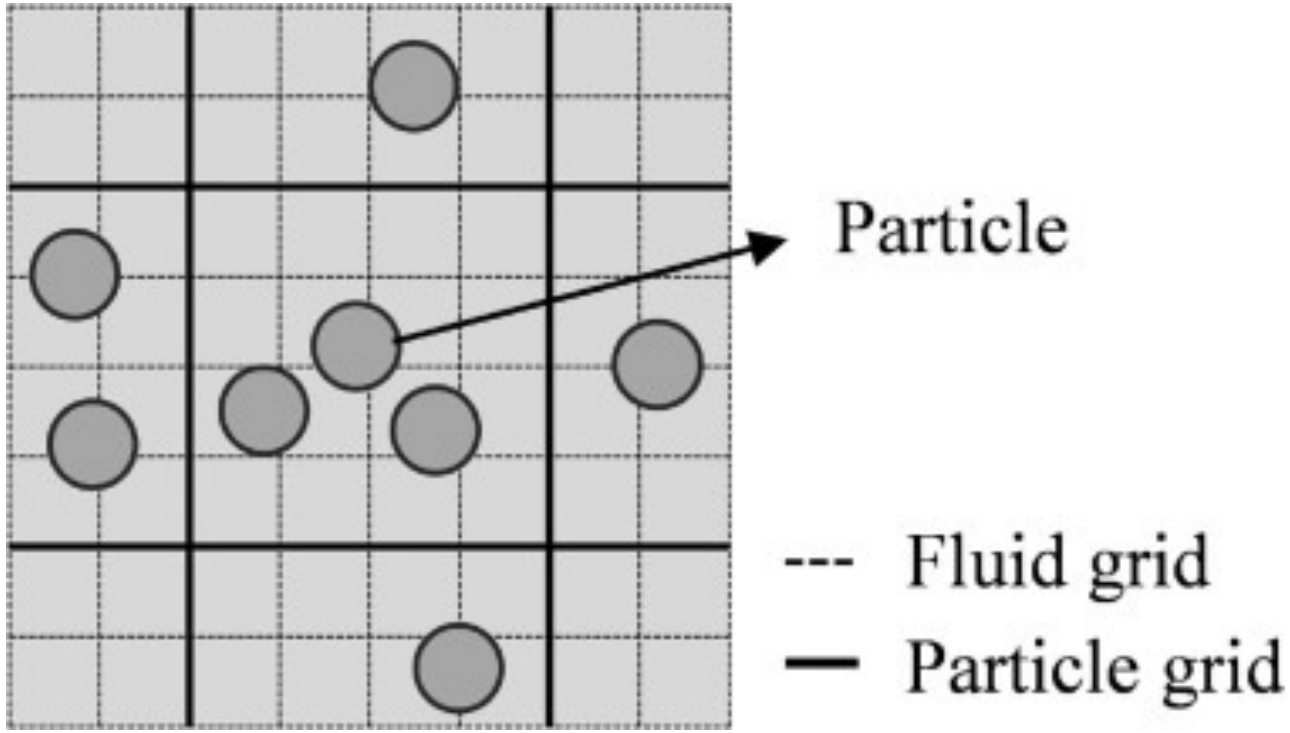




Classification relevant for parallelization



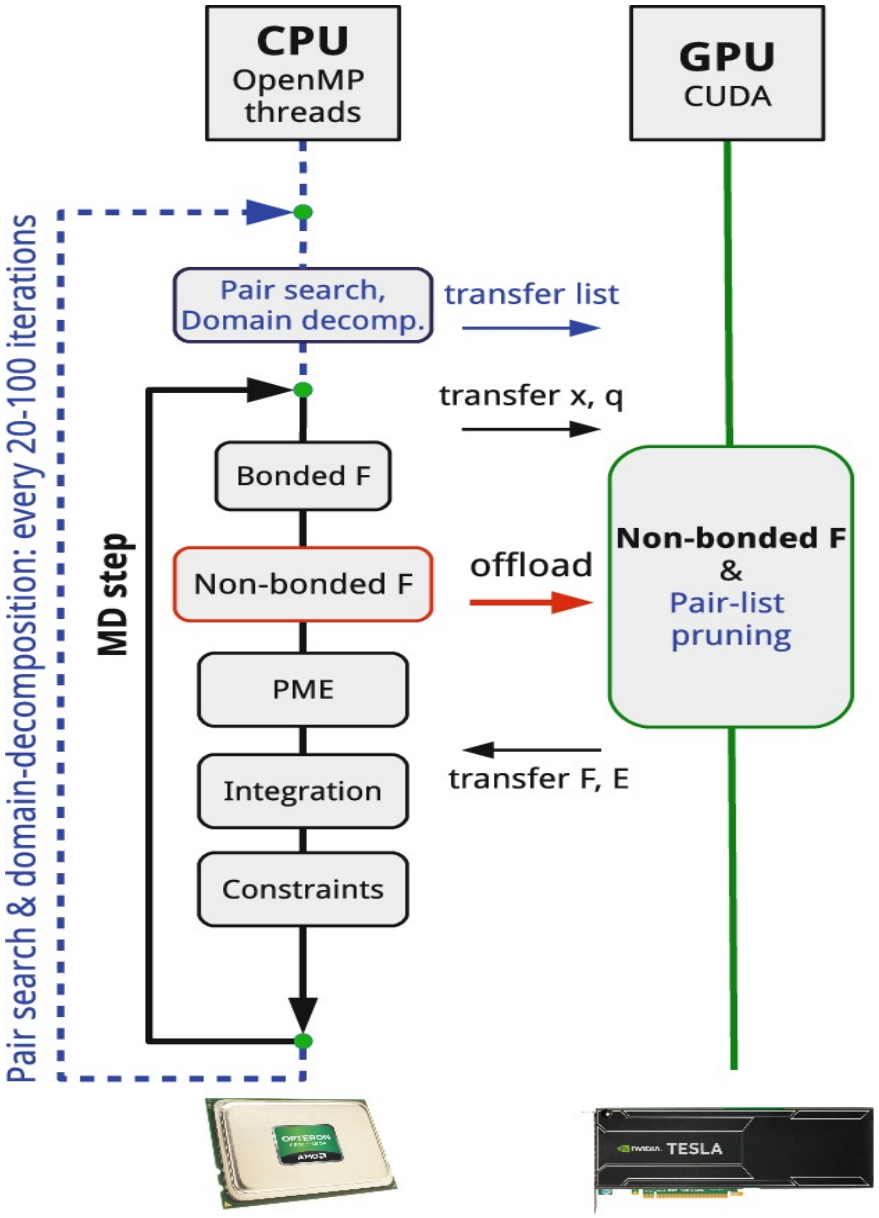
Purely Lagrangian



Lagrangian and Eulerian

Algorithm

- Initiation (accompanied with domain decomposition, will be explained later)
- Applying boundaries/injection if needed
- Creation of a cell list, neighborlist
- Force calculation
- Collision detection (if done separately)
- Translation, rotation OR integration
- Back to the top

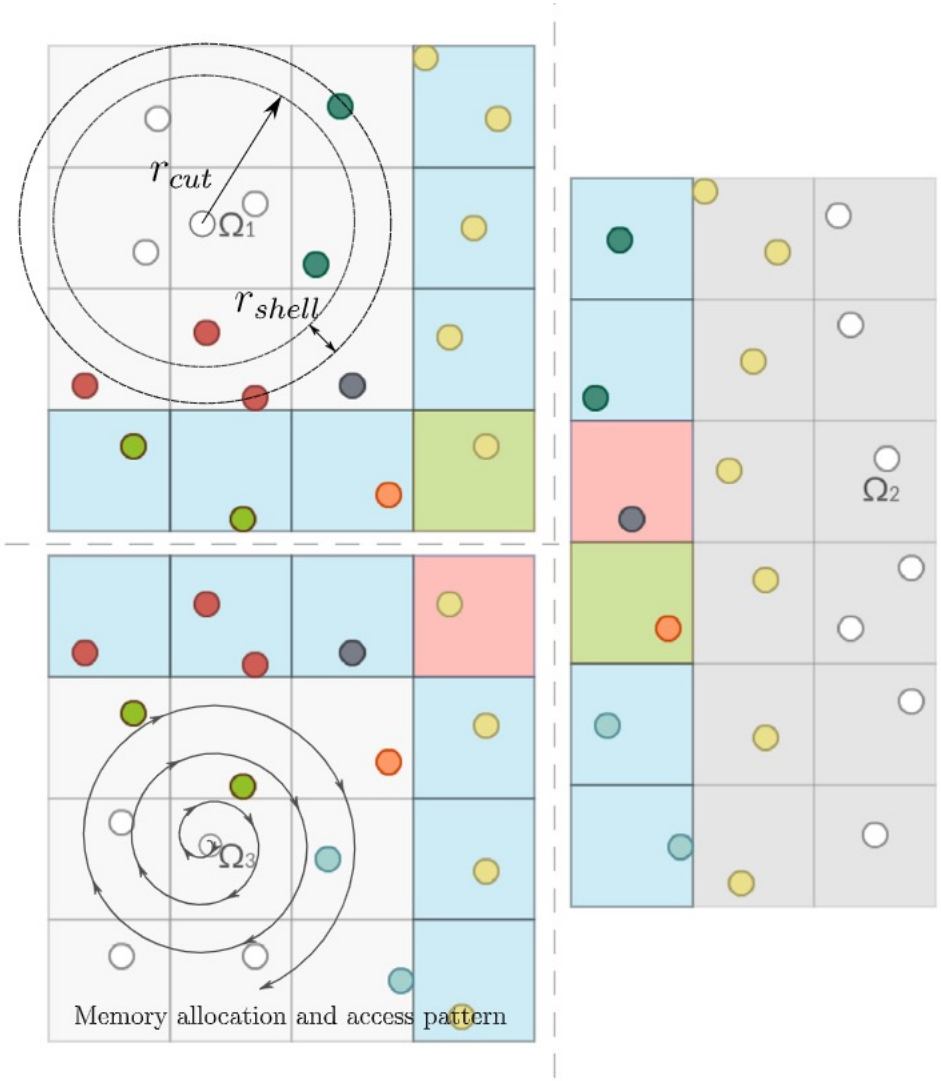


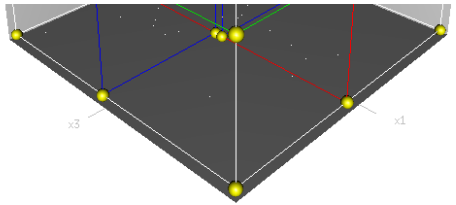
Single core optimization

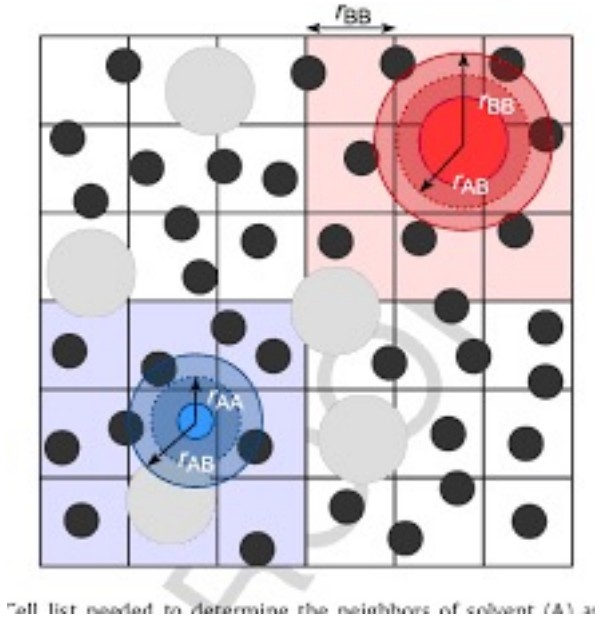
- A big chunk of efficiency depends on efficiency of the single core version.
- Particle based techniques are NOT memory bound but are compute bound.
- If your PB simulation is proving to be memory bound then you are missing a lot (cache)!
- How?
  - Use of efficient Data structures
  - Cache optimization
  - Vectorization (or even off load some parts to GPUs)

Data struc

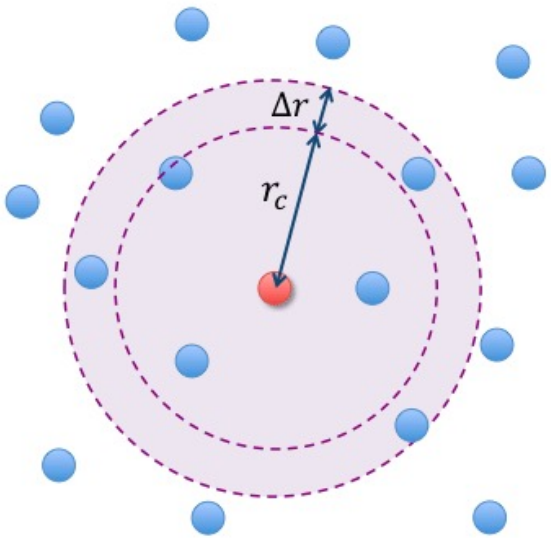
- Cell list
  - Kd-tree
  - Stencil volu
- Neighbour lis
  - Verlet list
- “Bonded “ en





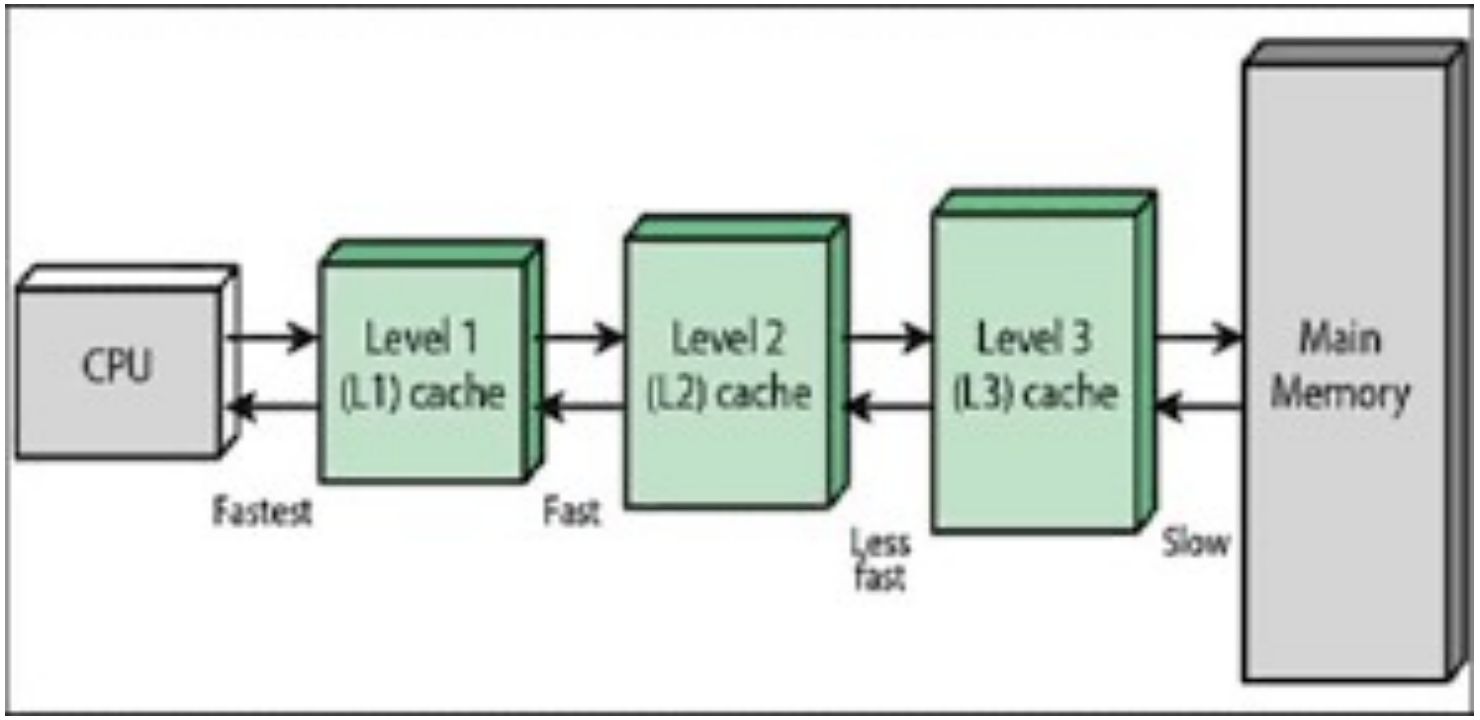


ised
fs)



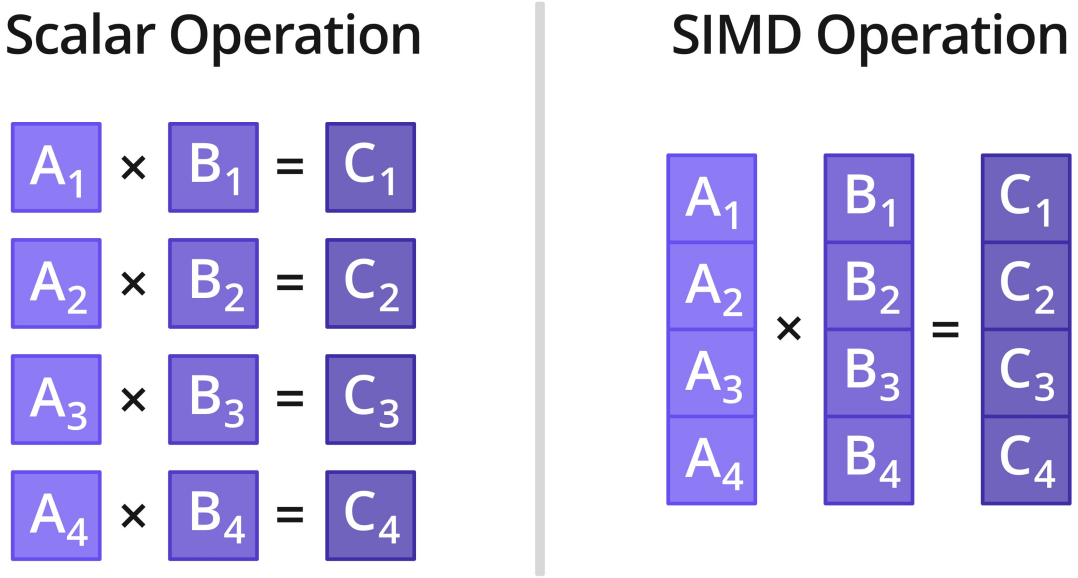
Cache Optimization

- Cache hierarchy
- Check access pattern of most compute intensive part of the program.
- In this case, they are nb-list query, force calculation, integration.



Vectorization

- Operations such as distance calculation, dot products etc. can be vectorized using SIMD.
- Results in maximum usage of the CPU registers and prevents unwanted repetitive calculations.
- Code can look a bit ugly.
- Off loading certain operations to accelerators/GPU is also an option.

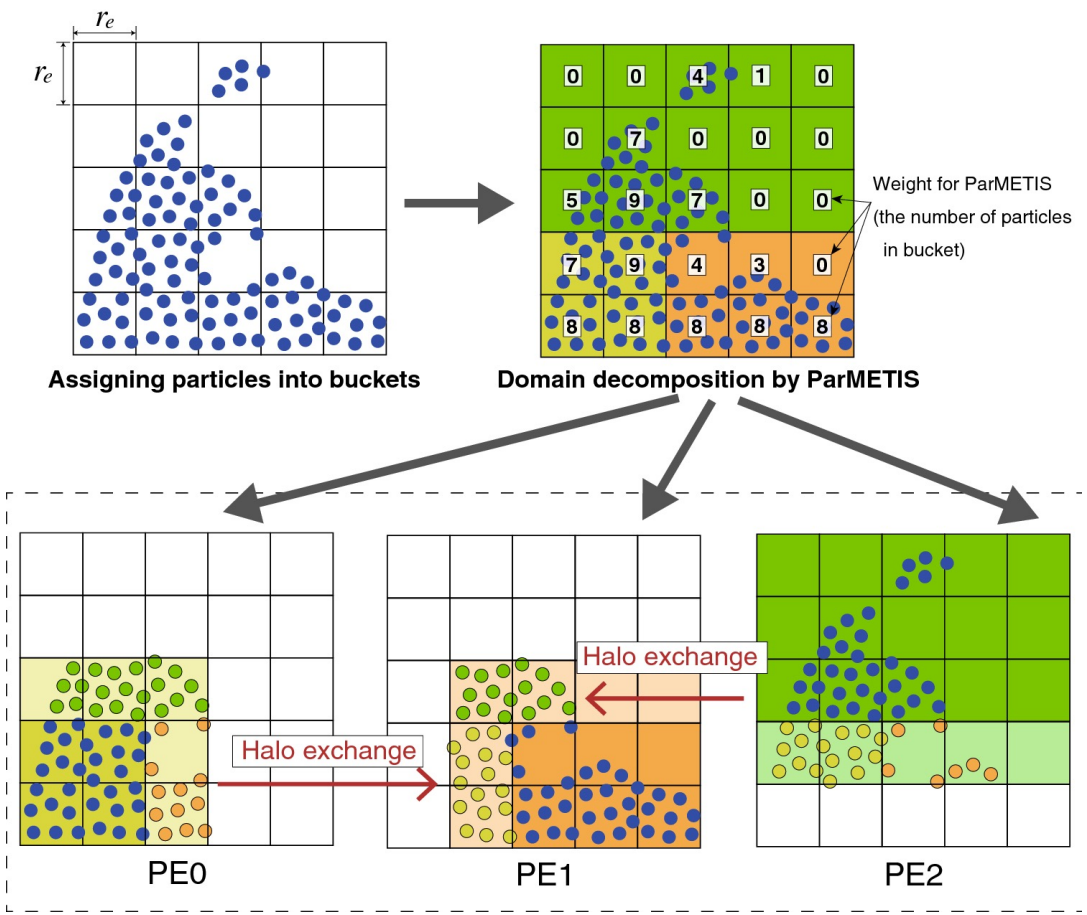


Heading towards multi-core

- Can be shared memory (OpenMP) or distributed memory (MPI) or both!
- Code conversion wise OpenMP is typically easier to try out.
  - Code still looks (almost) the same.
  - All information is still accessible (shared memory).
  - Easier to debug.
  - Lesser cache is available per thread.
  - Parallelism is typically ID based.
- MPI needs to start with a thought process ☺
  - Code may not look the same after.
  - All information is not available any more.
  - Errors look scary and is harder to debug as well.
  - A whole physical core is executing a process.
  - Parallelism is typically based on sub-domains.

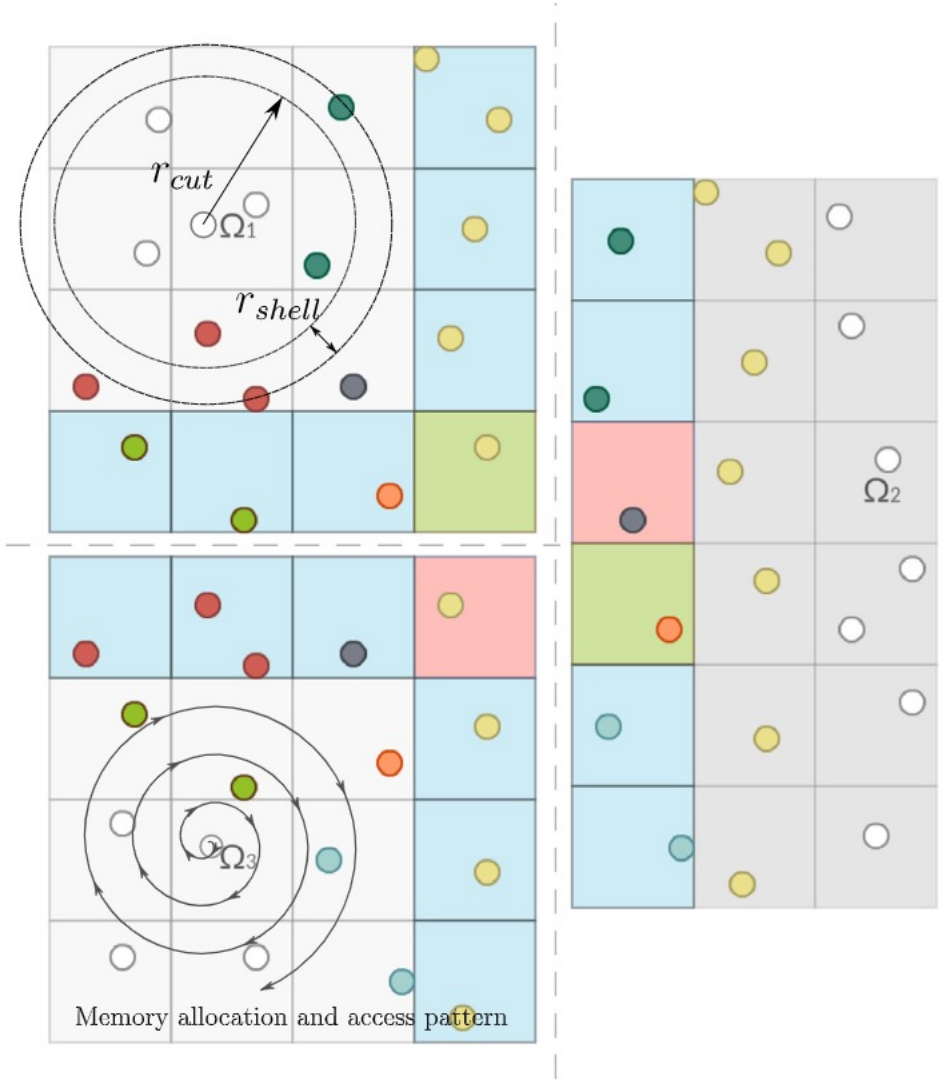
Domain Decomposition

- Can be static or dynamic.
- Depends on type of simulation.
  - A static grid based (Eulerian) system will perform better on a static grid.
  - Dynamic decomposition required for LOAD BALANCING.
  - A simulation with both static grid and dynamic entities need best of both worlds but this needs to be tested.
- Do not forget the inlet and the outlet of the system.
  - Managing inlet of new entities can be complex once the domain is decomposed.
- Choose the right MPI topology for an appropriate decomp.



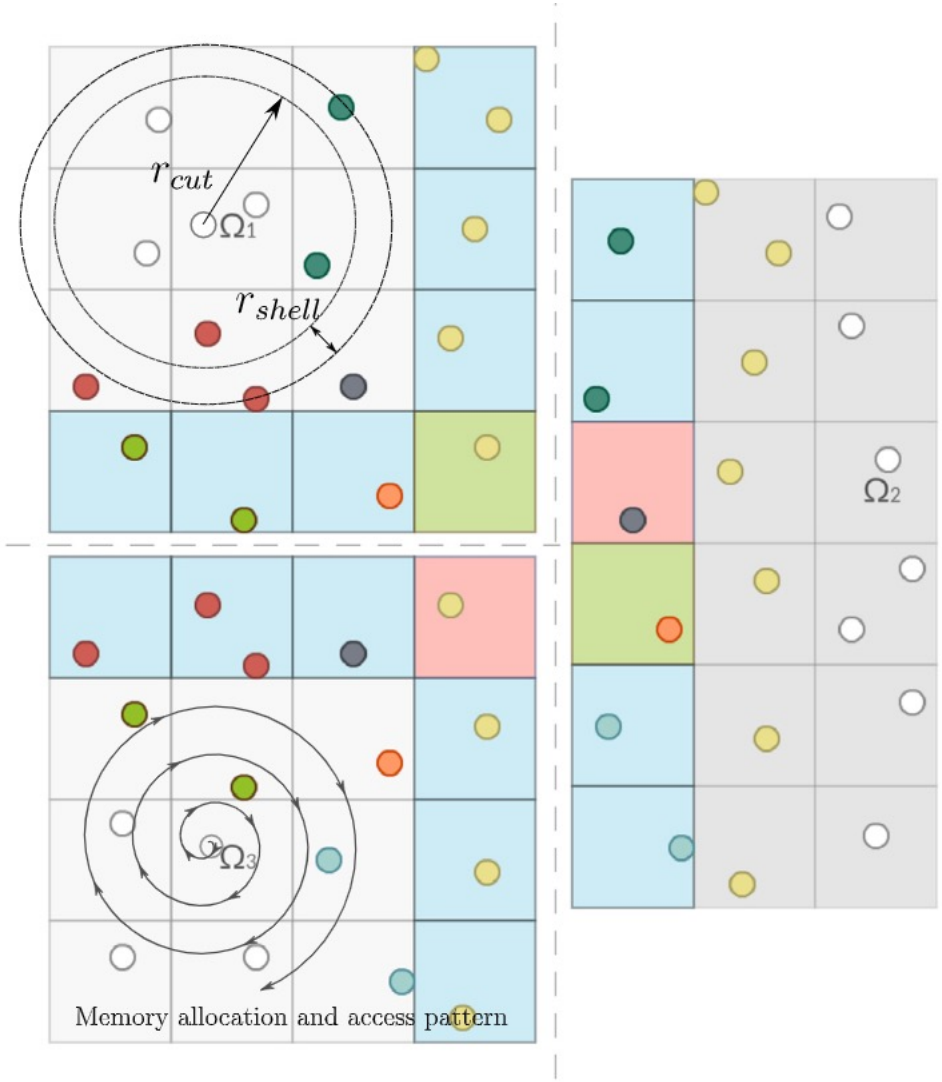
Local is better

- Coming back to the algorithm:
  - Can my algorithm run locally on each sub-domain?
  - Which parts need communication?
  - What kind of communication is required and where?
    - Just with the nbrs.
    - All to one or one to all
  - What does it mean for the overall performance?
- DBM and DSMC example
  - One algorithm relies on time
  - The other relies on a random partner.



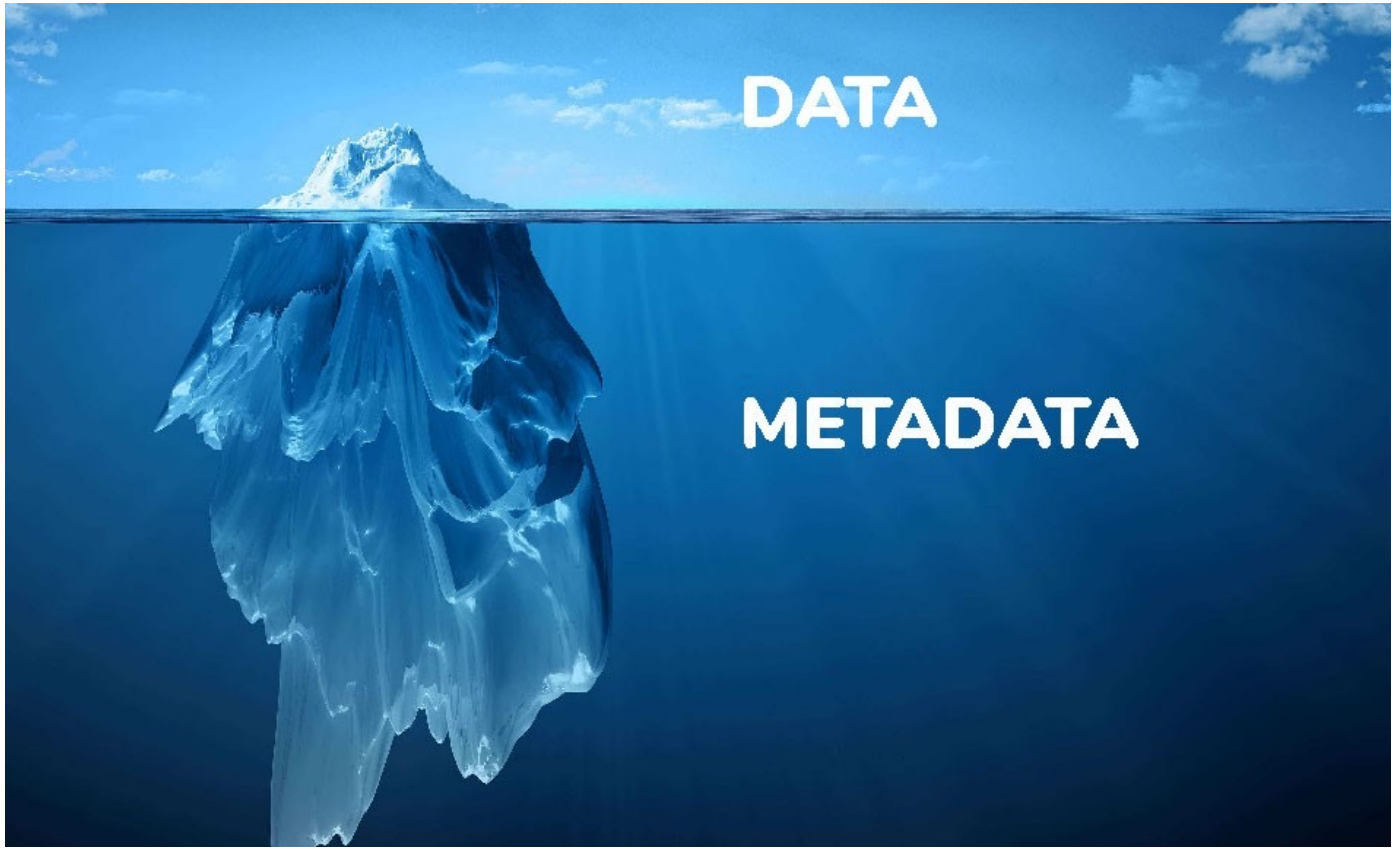
Do I see the whole picture?

- The short and correct answer is NO. ☺
- Is my algorithm the same at edges as in the bulk?
  - Does this require extra communication?
- How many layers of Halo regions will I need?
  - The more you need, the larger data size you need to communicate.



Serialization considerations

- Whenever I need to communicate, I need to serialize the data and deserialize it on the other side.
- Of course, this costs extra computation, therefore it is better to use easily mutable data structures and some metadata.
- Make it easier for yourself and the computer to de-serialize the data.
- The whole game is about access here, therefore think about cache and as far as possible do your operations in blocks.



Synchronization

- No algorithm is perfectly parallel.
- Some steps in the algorithm will depend on other steps to finish in the nearby processes.
- Lesser in-sync the better.

- Initiation (accompanied with domain decomposition, will be explained later)
- Applying boundaries/injection if needed
- Creation of a cell list, neighborlist
- Force calculation
- Collision detection (if done separately)
- Translation, rotation OR integration
- Back to the top

Programming tips - Approach

- It is a no brainer to start with a small representative system.
- It is also good to make sure that the representative system rigourously tests your algorithm.
- MPI typically requires some re-structuring of your original code, therefore take it slow initially. (Use #ifndef and #ifdef)
- Start with I/O first!! ☺ ⟶ Visualization is very important!
- Every process will have its own mind!! ☺
- Create helper functions that dump out data with which you can debug.

Unit tests and documentation

- In a nutshell → REQUIRED!
- What are unit tests?
- Create unit tests for literally every function.
- Create a documentation pipeline during the starting stages of any project. (Doxygen is an example)
- Constantly check for memory leaks and warnings during the coding stage (Valgrind is your friend. ☺)

Why is my program failing?

- Memory errors such as violations, race conditions.
- MPI Dead locks.
- Logical errors.
- Sudden performance reductions.
- I/O violations.

Parallel debugging is an Art

- Debugging can be cumbersome, therefore unit tests are very important.
- Parallel debuggers (apart from gdb and valgrind) are a boon BUT they need not point always to the right place.
- This is where the helper functions that dump data REALLY help!
- Profile your code from time to time to understand the bottle necks, these can also help find bugs at unexpected places.

Happy coding and parallelizing!! ☺