INCOMPRESSIBLE

DERIVATION OF THE U-ω EULER EQUATION FROM THE U-P FORM (BLACKBOARD)

Euler equations in velocity-pressure form

① $\frac{Du}{Dt} = \frac{\partial u}{\partial t} + (u\cdot\nabla)u = -\frac{1}{\rho}\nabla p$ (Momentum equation)

② $\nabla\cdot u = 0$ (Continuity equation — incompressibility)

$\nabla\times$①:

$$\nabla\times\frac{\partial u}{\partial t} + \nabla\times\left[(u\cdot\nabla)u\right] = \nabla\times\left(-\frac{1}{\rho}\nabla p\right)$$

$\nabla\times\left(-\frac{1}{\rho}\nabla p\right) = 0$ (i) (curl of gradient of a scalar field)

$$\frac{\partial\omega}{\partial t} + \nabla\times\left[\frac{1}{2}\nabla(u\cdot u) - u\times(\nabla\times u)\right] =$$

$\frac{1}{2}\nabla(u\cdot u)$: 0 (i)

$\nabla\times u = \omega$ (ii)

ω (ii)

(ii) $\nabla\times u = \omega$

$\frac{1}{2}\nabla(u\cdot u) = (u\cdot\nabla)u + u\times(\nabla\times u)$

$$= \frac{\partial\omega}{\partial t} + (u\cdot\nabla)\omega - (\omega\cdot\nabla)u + \omega(\nabla\cdot u) - u(\nabla\cdot\omega) = 0$$

$\omega(\nabla\cdot u)$: ②

$\nabla\cdot(\nabla\times\omega) = 0$

$$\Rightarrow \boxed{\frac{\partial\omega}{\partial t} + (u\cdot\nabla)\omega = (\omega\cdot\nabla)u}$$

Further simplification in 2D: $(\omega\cdot\nabla)u = 0$

$$\Rightarrow \boxed{\frac{\partial\omega}{\partial t} + (u\cdot\nabla)\omega = 0}$$ 2D

RECOVERING OF VELOCITIES FROM VORTICITIES

$\nabla\times\left(\omega = \nabla\times u\right)$

$$\nabla\times\omega = \nabla\times(\nabla\times u) = \nabla(\nabla\cdot u) - \nabla^2 u$$

$\nabla(\nabla\cdot u)$: 0 ②

$$\Rightarrow \boxed{\Delta u = -\nabla\times\omega}$$

Poisson Equation

OR

$$\omega = \nabla\times u = \nabla\times(\nabla\times\psi)$$

$$= \nabla(\nabla\cdot\psi) - \nabla^2\psi$$

$\nabla(\nabla\cdot\psi)$: 0 (2D)

$$\Rightarrow \boxed{\Delta\psi = -\omega}$$

Poisson Equation

(scalar)

$$\rightarrow u = \nabla\times\psi$$

Green's Function Solution for the Diffusion Equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \qquad (1D)$$

the Green's function solution satisfies

$$\left[D\frac{\partial^2}{\partial x^2} - \frac{\partial}{\partial t}\right] G = -\delta(x)\delta(t)$$

the Fourier Transform $\hat{G} = \int_{-\infty}^{\infty} G e^{-ikx} dx$

yields:

$$\left[Dk^2 + \frac{\partial}{\partial t}\right] \hat{G} = \delta(t)$$

this is now an ODE which has solution

$$\hat{G} = H(t) e^{-tDk^2} \qquad \left[\begin{array}{l} H: \text{Heaviside function} \\ \frac{\partial H(t)}{\partial t} = \delta(t) \end{array}\right]$$

test:

$$\frac{\partial \hat{G}}{\partial t} = \delta(t) - Dk^2 e^{-tDk^2}$$

Taking the inverse FT of $\hat{G}$, we obtain

$$G = \frac{1}{2\pi}\int_{-\infty}^{\infty} H(t) e^{-tDk^2} e^{ikx} dk =$$

$$= \frac{H(t)}{2\pi}\int_{-\infty}^{\infty} e^{-tD\left(k^2 - \frac{ikx}{tD}\right)} dk =$$

$$= \frac{H(t)}{2\pi}\int_{-\infty}^{\infty} e^{-tD\left(k - \frac{ix}{2tD}\right)^2 - \frac{x^2}{4tD}} dk =$$

$$= \frac{H(t)}{2\pi} e^{-\frac{x^2}{4tD}} \int_{-\infty}^{\infty} e^{-tD\left(k - \frac{ix}{2tD}\right)^2} dk =$$

$$\int_{-\infty}^{\infty} e^{-ak^2} dk = \sqrt{\frac{\pi}{a}}$$

$$= \frac{H(t)}{\sqrt{4\pi tD}} e^{-\frac{x^2}{4Dt}} \Longrightarrow \text{Gaussian!}$$

→ we are interested in $t > 0$, thus

$$G = \frac{1}{\sqrt{4\pi tD}} e^{-\frac{x^2}{4Dt}}$$

This is a short range interaction!

Cell-Lists

High Performance Computing for Science and Engineering

CSE**lab**
Computational Science & Engineering Laboratory
http://www.cse-lab.ethz.ch

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diffusion as N-Body Problem

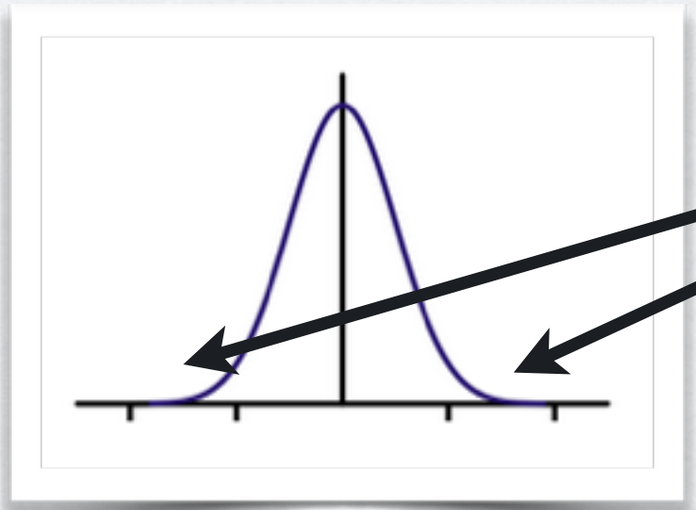
The fundamental solution to the diffusion equation

$$\frac{\partial u}{\partial t} = D\Delta u$$

is given by the Green's function

$$G(\mathbf{x}, t) = \left(\frac{1}{4\pi Dt}\right)^{d/2} e^{-\frac{\mathbf{x}\cdot\mathbf{x}}{4Dt}}$$

which is a *d*-dimensional Gaussian function



fast decay

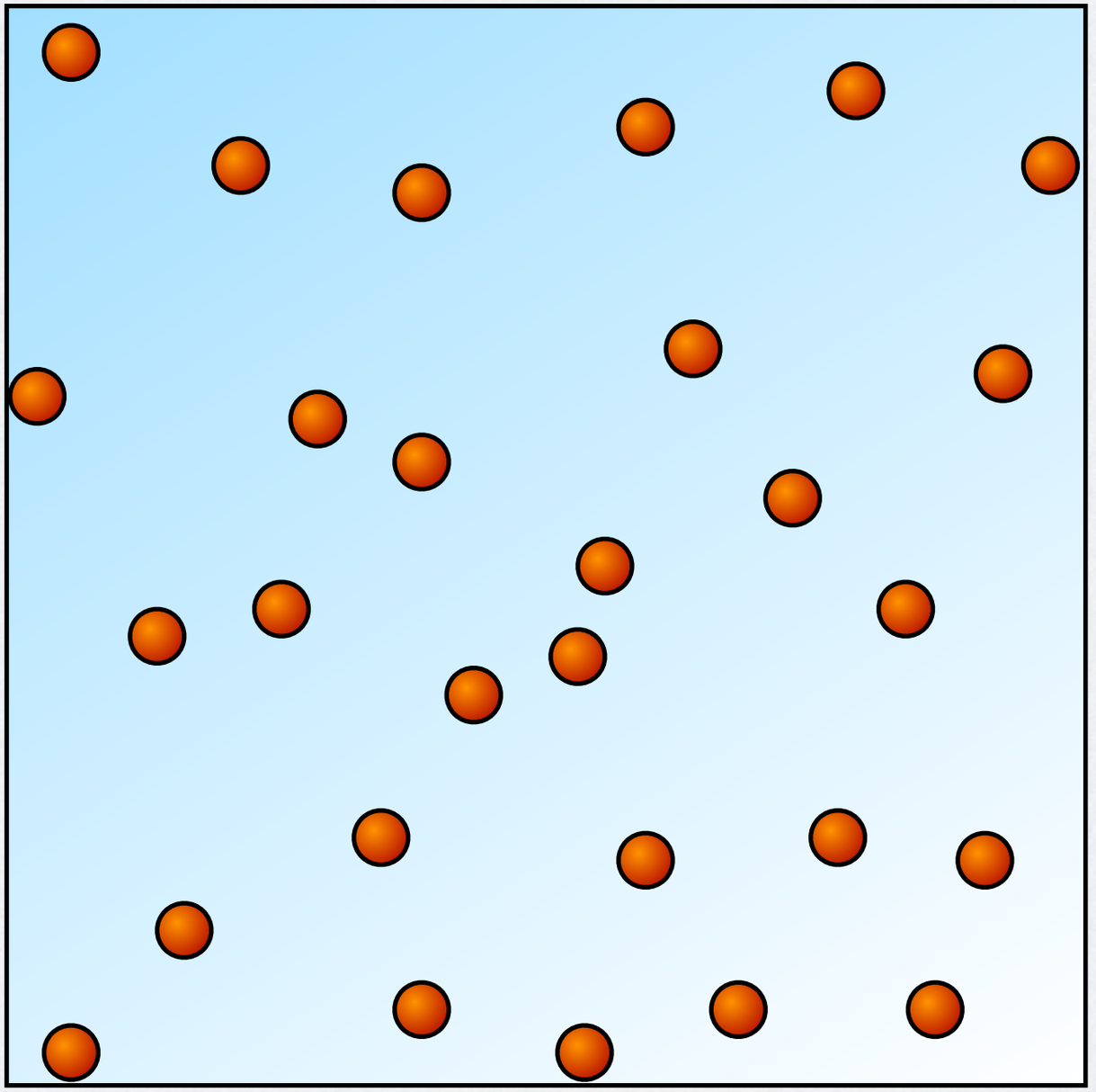
short-range interactions

N-Body Problem in MD

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i$$

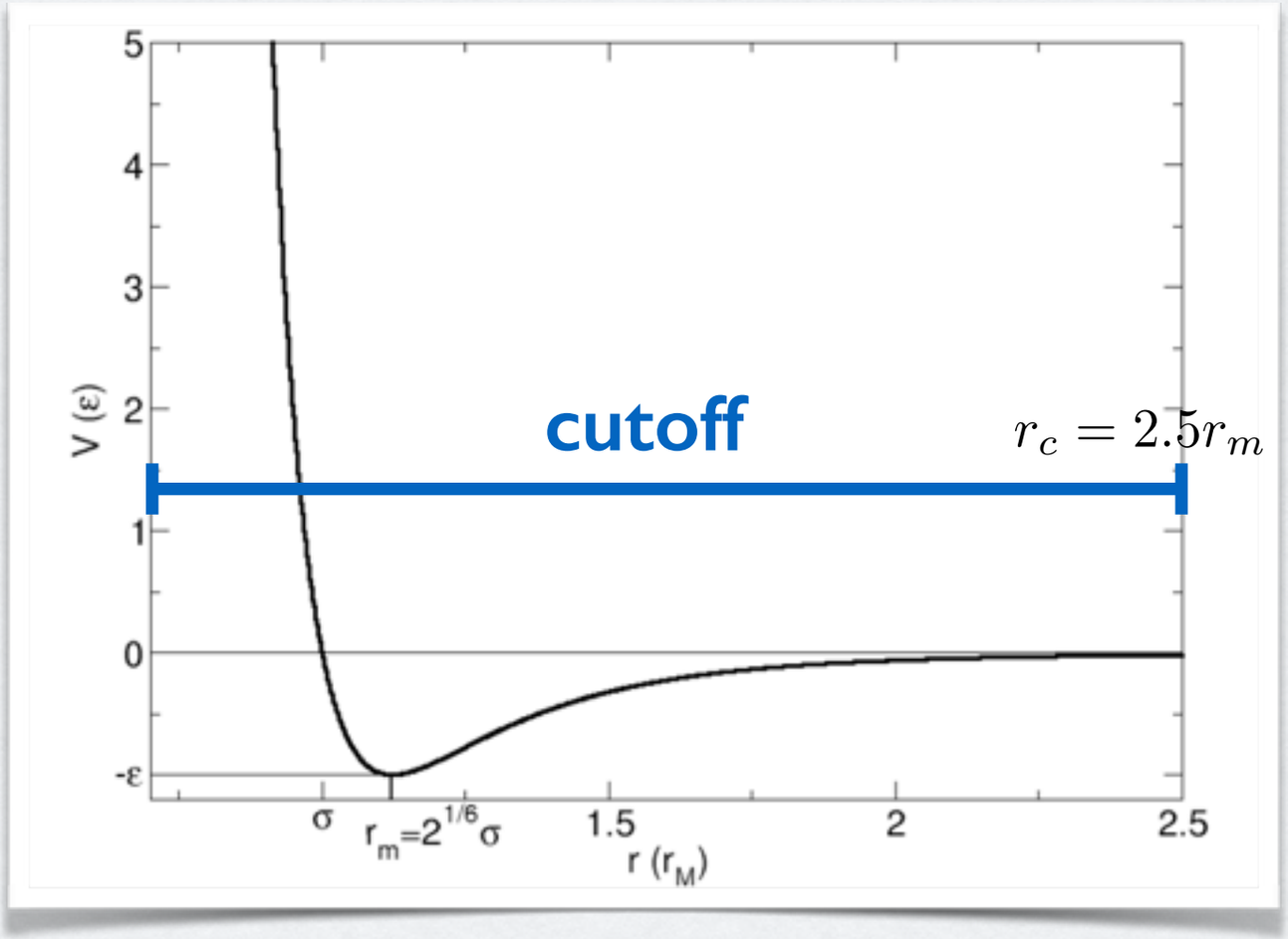
$$m_i \frac{d\mathbf{u}_i}{dt} = \sum_{j=1;j\neq i}^{N} \mathbf{F}_{ij}$$

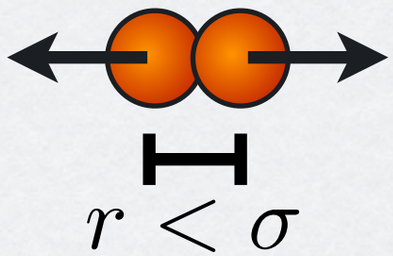
$$\mathbf{F}_{ij} = -\frac{\mathbf{x}_i - \mathbf{x}_j}{||\mathbf{x}_i - \mathbf{x}_j||}\frac{dV}{dr}$$



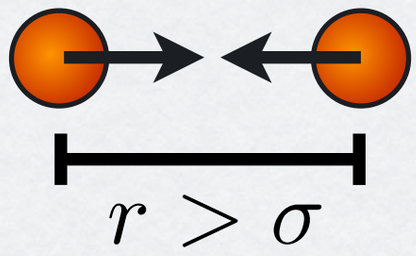
The Lennard-Jones Potential

$$V_c(r) = \begin{cases} 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] - V(r_c) & \text{for } r < r_c, \\ 0 & \text{else,} \end{cases}$$



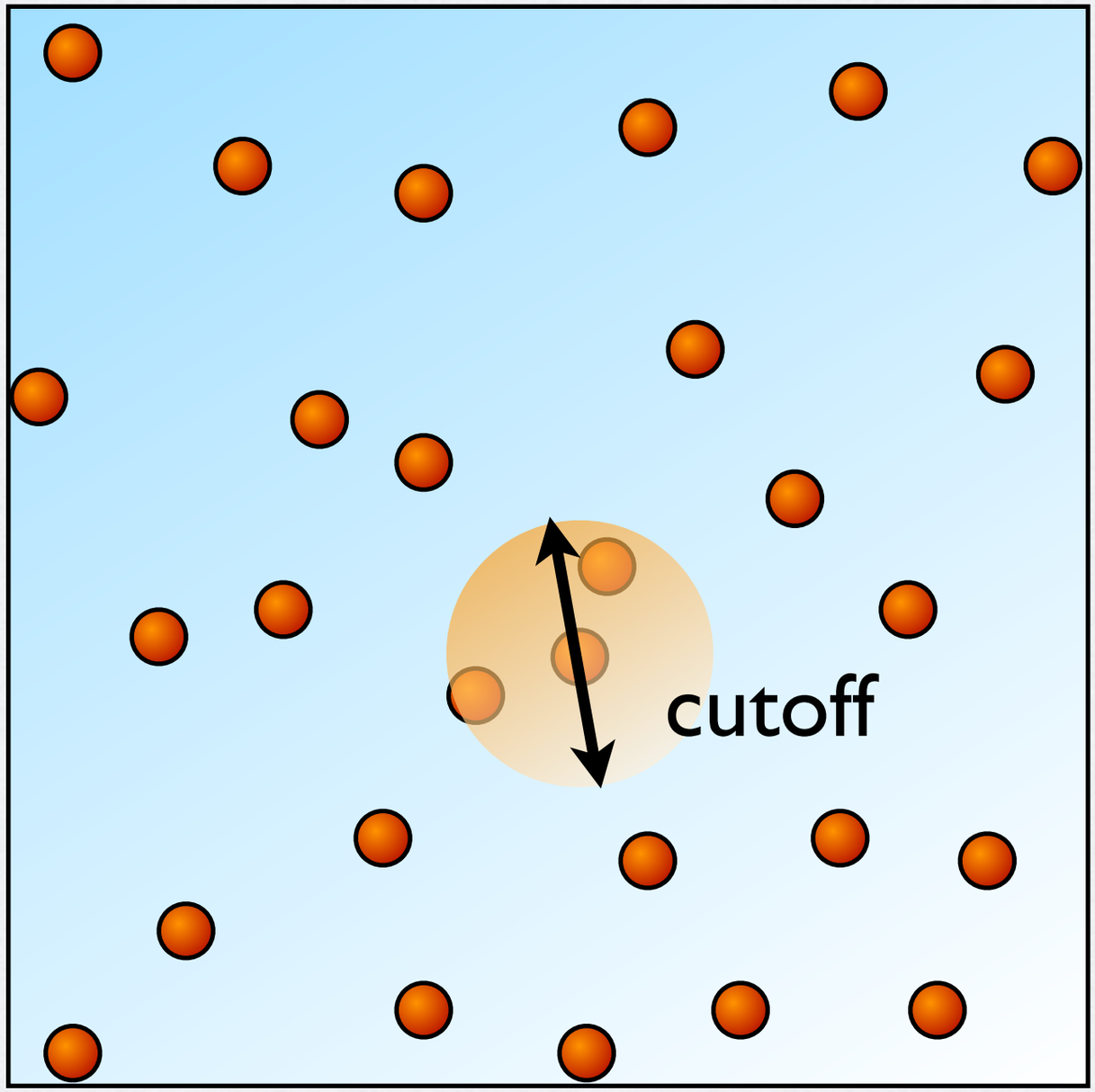


$r < \sigma$



$r > \sigma$

Local Interactions



Can we take advantage of this to avoid $\mathcal{O}(N^2)$ operations?

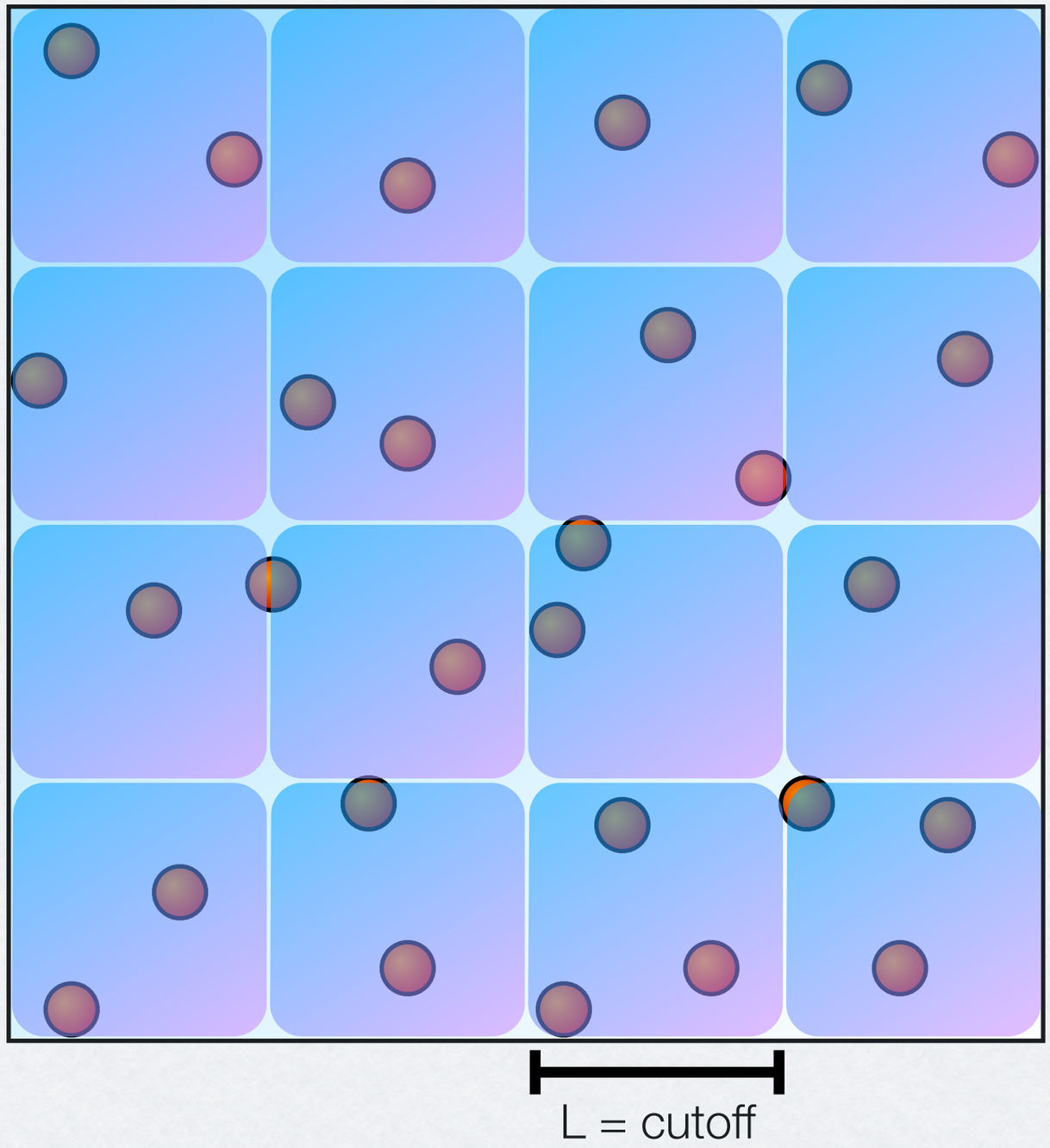
Idea - Subdivide Domain

Domain subdivision in *cells:*

- Cell size is the cutoff
- Cells contain particles

Particles:

- Spread into the cells according to their position



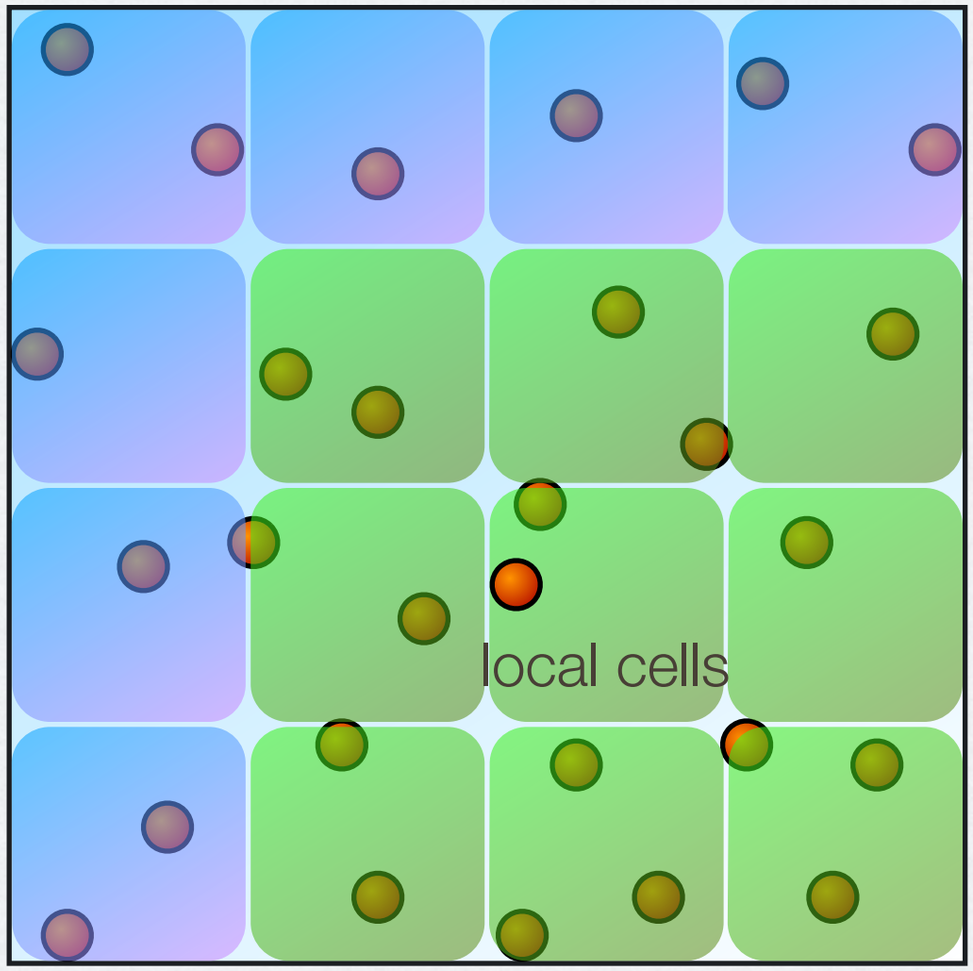
Computing Interactions

Interactions: only local cells contribute

Cost of N interactions (2D):

- c=N/m (average #particles/cell)
- assume constant c w.r.t. N
- cost for one particle: 9c
- total: 9cN ➡ $\mathcal{O}$(N)
- algorithmic improvement:

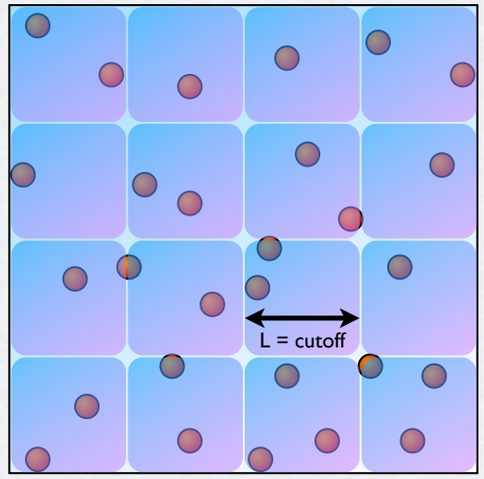
$$\frac{N^2}{9cN} = \frac{N}{9c}$$



m cells

Cell List Construction

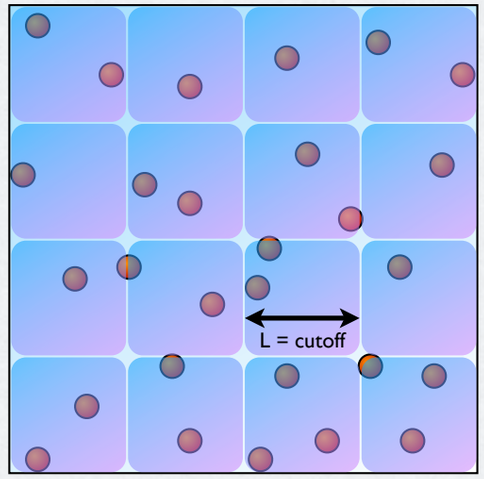
HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?



Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?

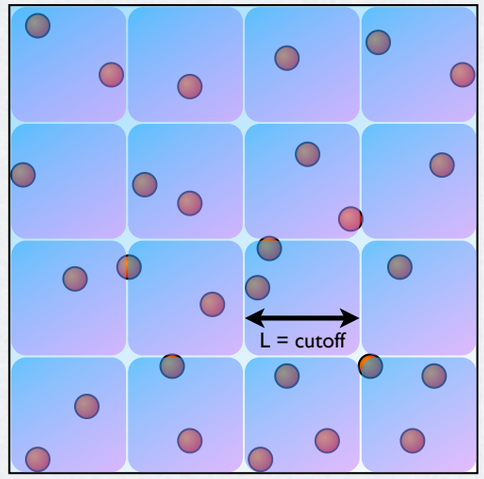
- Group particles according to the “address” of their cell



Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?

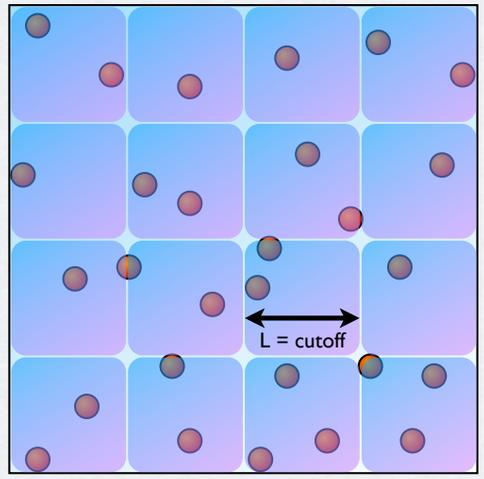
- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)



Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?

- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)
- Insertion of a particle in its associated cell in O(1)

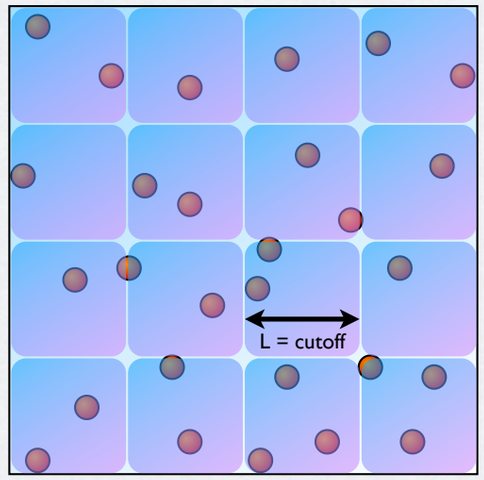


Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?

- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)
- Insertion of a particle in its associated cell in O(1)

➔ Cost for the creation of cell lists for N particles is in O(N)

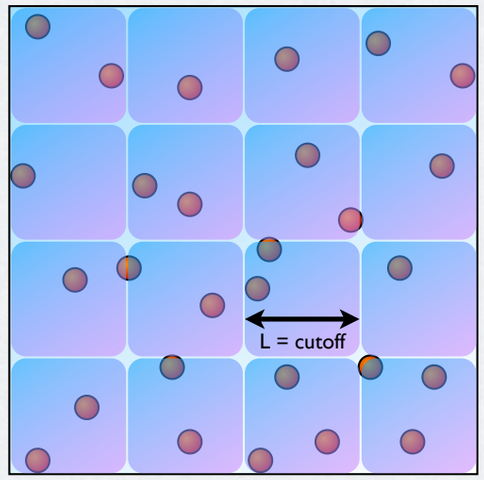


Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?

- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)
- Insertion of a particle in its associated cell in O(1)

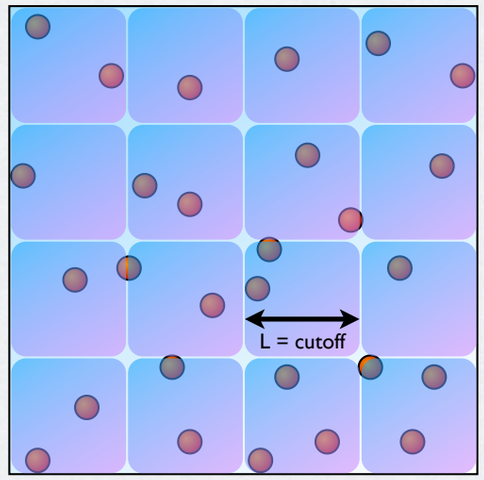
➔ Cost for the creation of cell lists for N particles is in O(N)



STL-based implementation with M^3 cells (3D, domain $[0,1)^3$)

Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?



- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)
- Insertion of a particle in its associated cell in O(1)

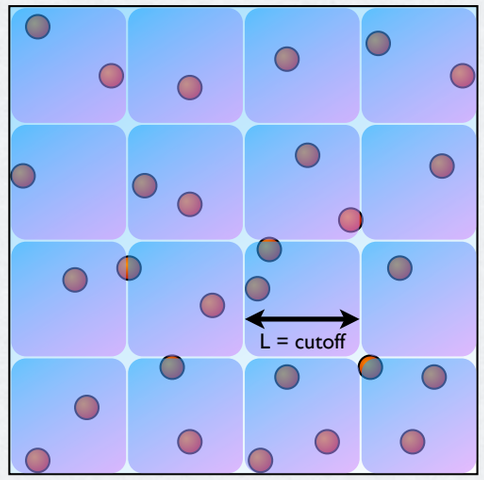
➔ Cost for the creation of cell lists for N particles is in O(N)

STL-based implementation with M^3 cells (3D, domain $[0,1)^3$)

- `std::vector<ParticleID*> cells(M*M*M)` contains a vector for each cell

Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?



- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)
- Insertion of a particle in its associated cell in O(1)

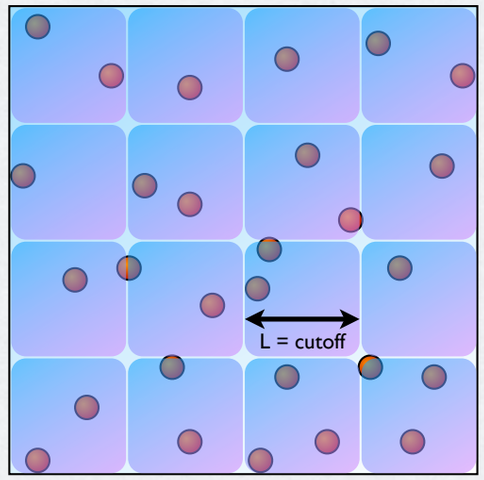
➔ Cost for the creation of cell lists for N particles is in O(N)

STL-based implementation with M^3 cells (3D, domain $[0,1)^3$)

- `std::vector<ParticleID*> cells(M*M*M)` contains a vector for each cell
- cell address for particle at (x,y,z): (ix,iy,iz) = (floor(x/L)), floor(y/L), floor(z/L)), L = 1/M

Cell List Construction

HOW TO FIND THE PARTICLE NEAREST NEIGHBORS?



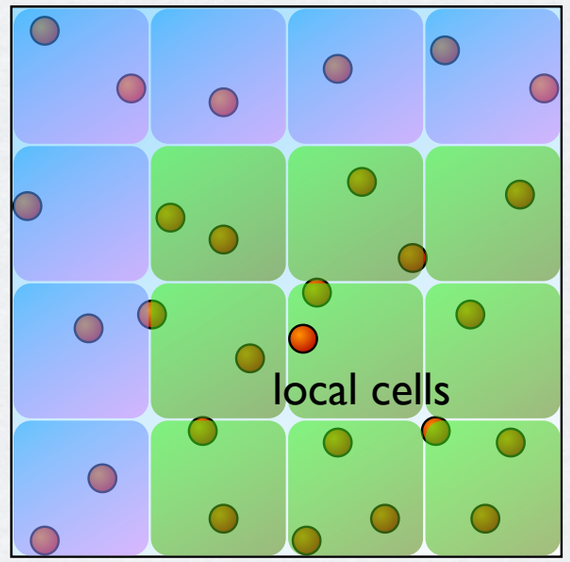
- Group particles according to the “address” of their cell
- Cell address for a single particle can be computed in O(1)
- Insertion of a particle in its associated cell in O(1)

➔ Cost for the creation of cell lists for N particles is in O(N)

STL-based implementation with M^3 cells (3D, domain $[0,1)^3$)

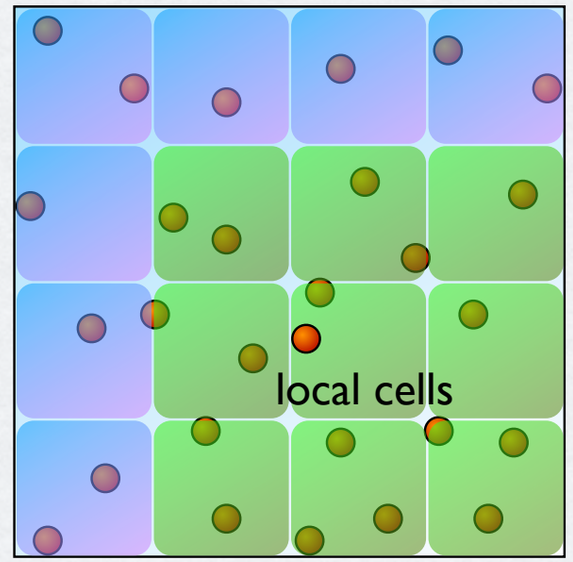
- `std::vector<ParticleID*> cells(M*M*M)` contains a vector for each cell
- cell address for particle at (x,y,z): (ix,iy,iz) = (floor(x/L)), floor(y/L), floor(z/L)), L = 1/M
  - 1D address into `cells`: `idx = ix + iy*M + iz*M*M = ix + M*(iy + M*iz)`

Cell List Traversal



Cell List Traversal

- Given an initialized `std::vector<ParticleID*> cells(M*M*M)`

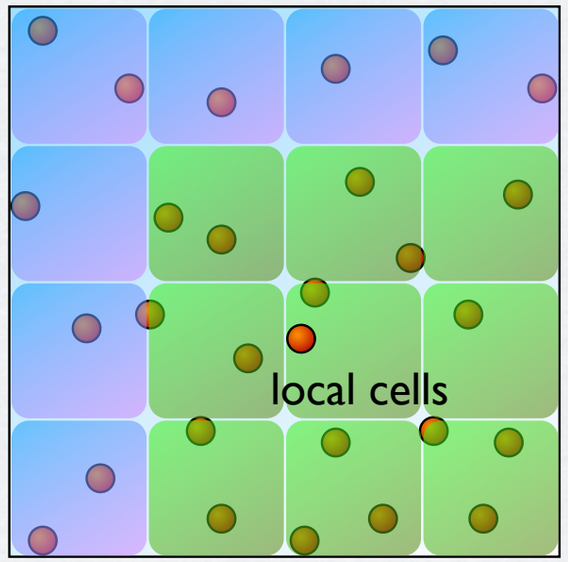


Cell List Traversal

- Given an initialized `std::vector<ParticleID*> cells(M*M*M)`

- **Algorithm**:

  Loop over all cells[i]



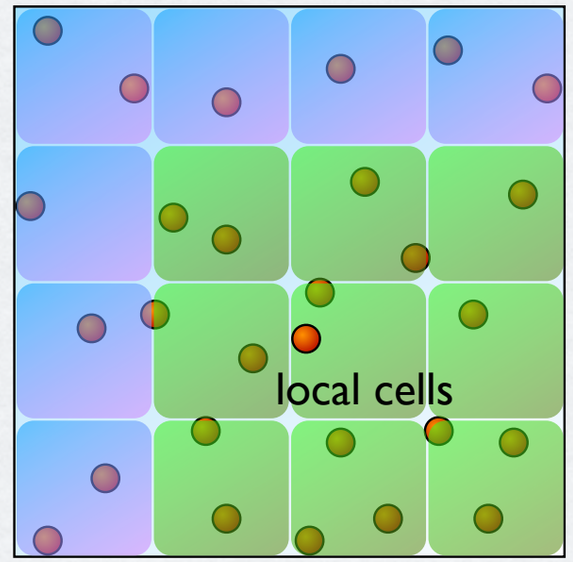
Cell List Traversal

- Given an initialized
  `std::vector<ParticleID*> cells(M*M*M)`

- **Algorithm**:

Loop over all cells[i]

- Loop over all neighbors cells[j] (incl. cells[i] itself)



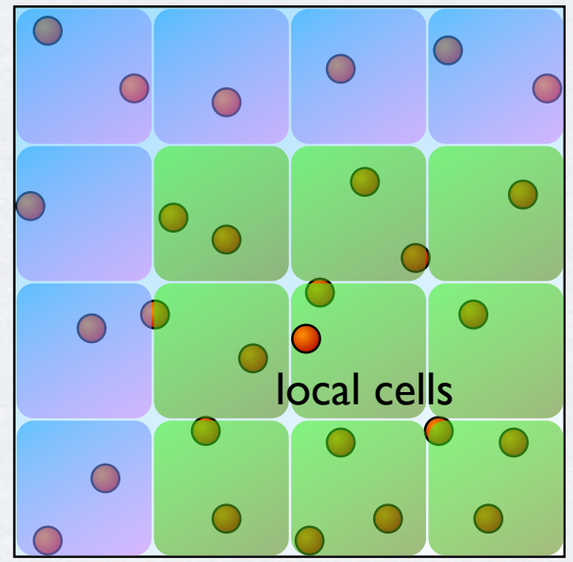
Cell List Traversal

- Given an initialized `std::vector<ParticleID*> cells(M*M*M)`

- **Algorithm**:

Loop over all cells[i]

- Loop over all neighbors cells[j] (incl. cells[i] itself)
  - Loop over all particles p[k] in cells[i]



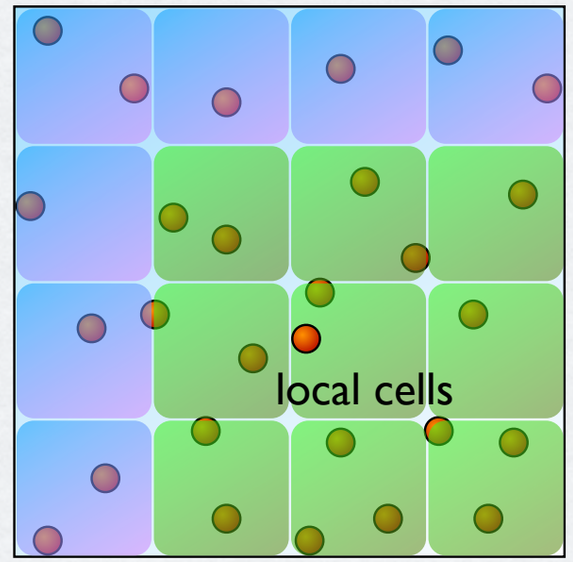
Cell List Traversal

- Given an initialized `std::vector<ParticleID*> cells(M*M*M)`

- **Algorithm**:

Loop over all cells[i]

- Loop over all neighbors cells[j] (incl. cells[i] itself)
  - Loop over all particles p[k] in cells[i]
    - Compute interaction of p[k] with all p[I] in cells[j] and update force of p[k]



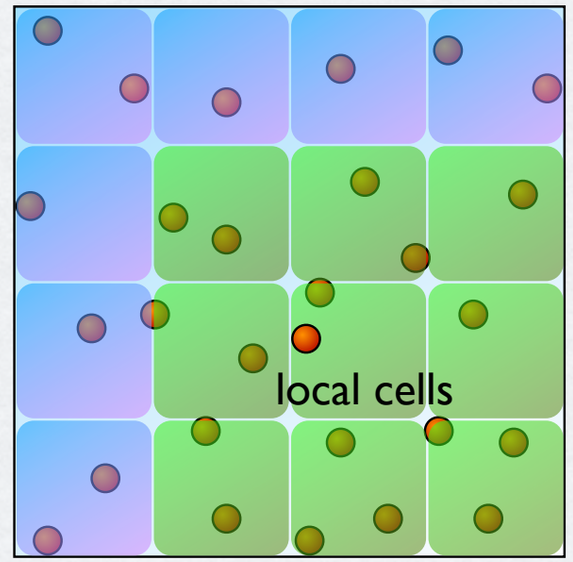
Cell List Traversal

- Given an initialized `std::vector<ParticleID*> cells(M*M*M)`

- **Algorithm**:

Loop over all cells[i]

- Loop over all neighbors cells[j] (incl. cells[i] itself)
  - Loop over all particles p[k] in cells[i]
    - Compute interaction of p[k] with all p[l] in cells[j] and update force of p[k]
      - Careful to skip interaction with own particle (i.e. p[k] == p[l])



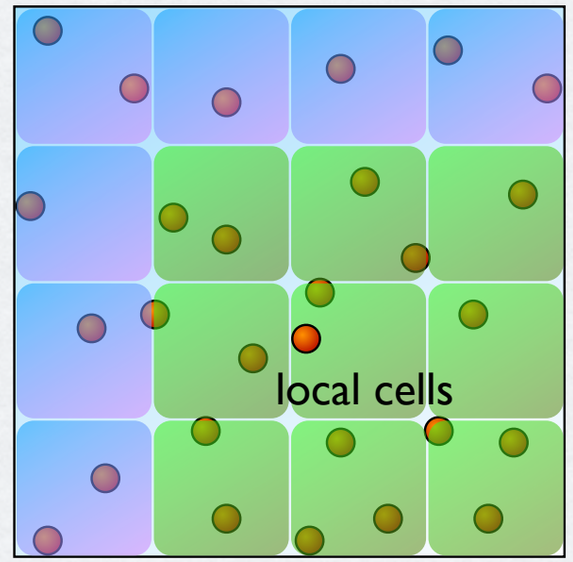
Cell List Traversal

- Given an initialized `std::vector<ParticleID*> cells(M*M*M)`

- **Algorithm**:

Loop over all cells[i]

- Loop over all neighbors cells[j] (incl. cells[i] itself)
  - Loop over all particles p[k] in cells[i]
    - Compute interaction of p[k] with all p[l] in cells[j] and update force of p[k]
      - Careful to skip interaction with own particle (i.e. p[k] == p[l])
      - Always check the cutoff distance: Cell list guarantees that all particles within the cutoff are in neighboring cells, not the other way around!

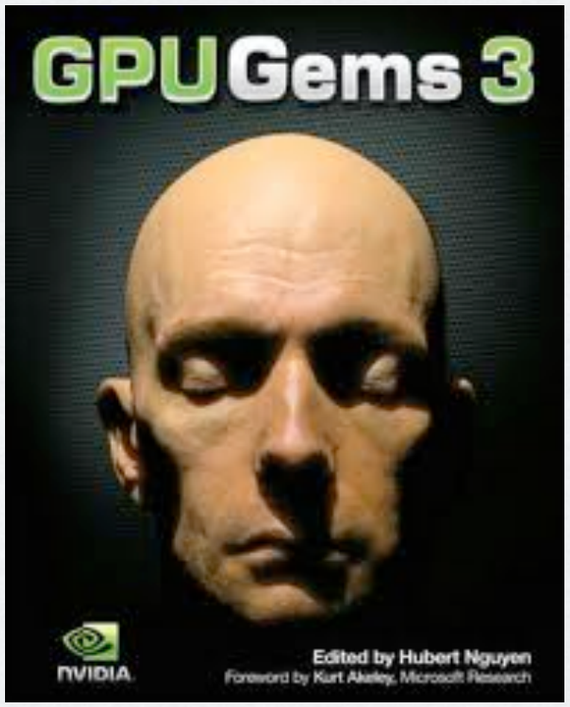


Using Cell List

- Cell list is created at each time-step
- Cell list creation is $\mathcal{O}$(N), it does not mean cheap!
- Particle position coherence can be exploited
  - Keep previous cell list
  - Refresh cell list:
    - Look for particles that moved into another cell
    - Erase the particle from the old cell
    - Insert the particle in the new cell

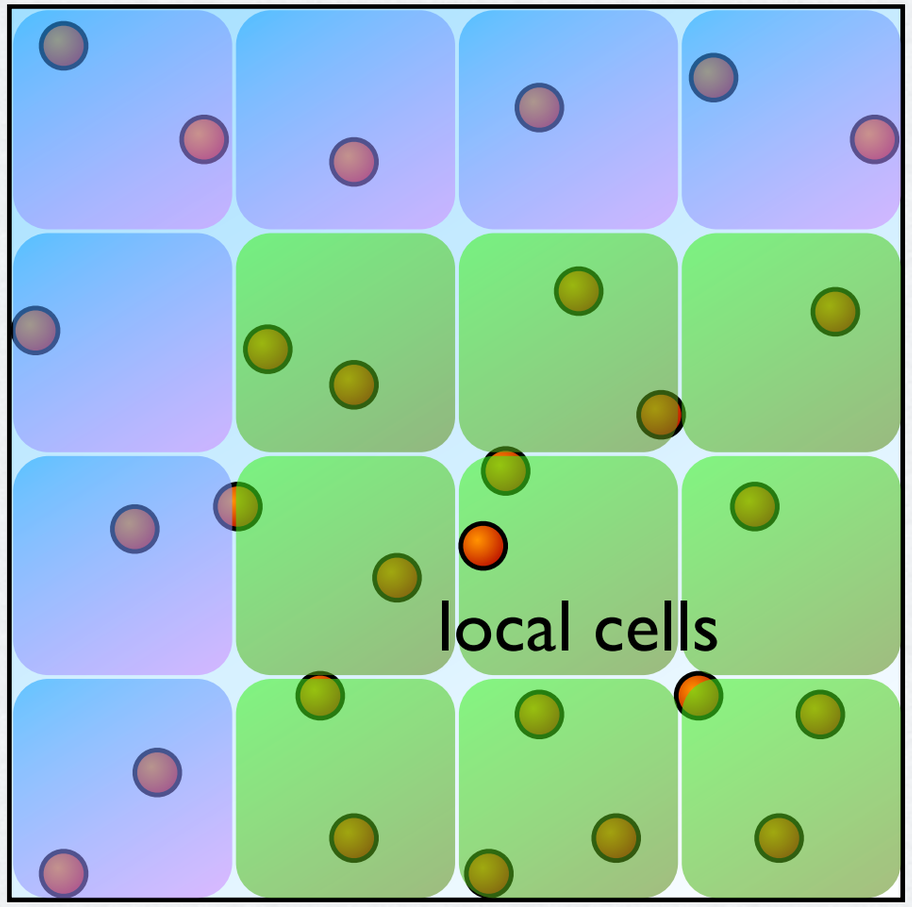
Cell List Implementation

- Information taken from GPU Gems 3
  - GPU Gems 3 is a collection of state-of-the-art GPU programming examples
  - available online: http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html



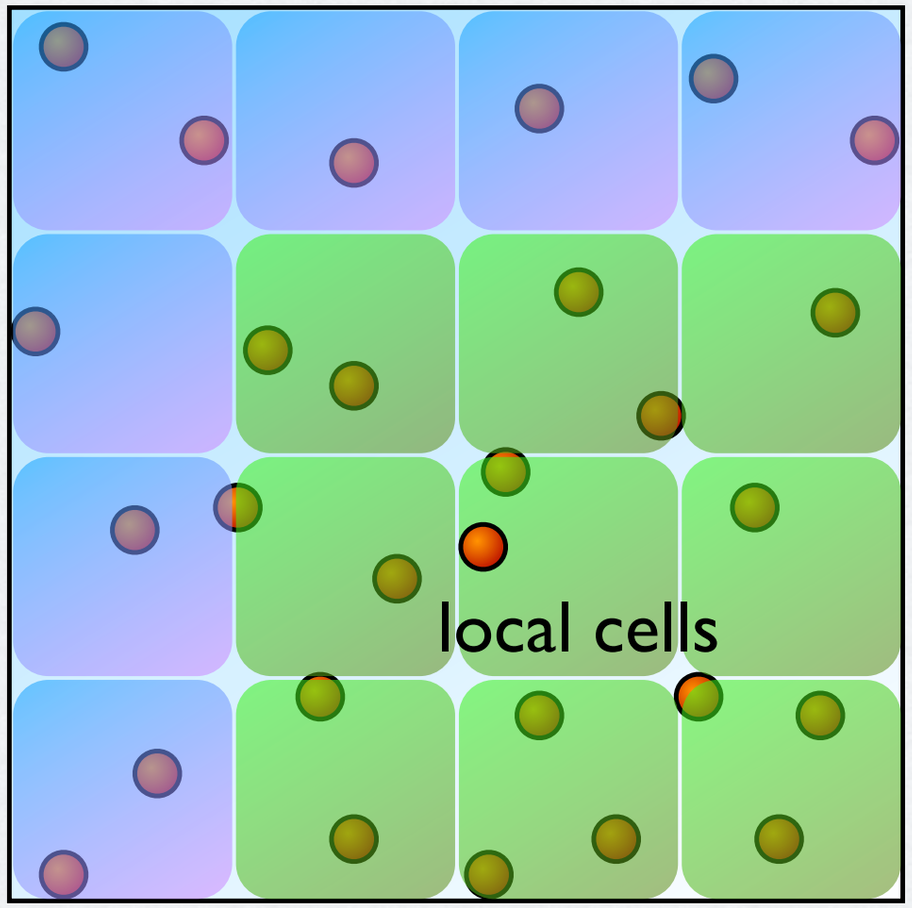
- Broad-phase collision detection with CUDA
  - called “**Spatial Subdivision**” algorithm
- Analogous to cell-lists in MD

Sequential Implementation



Sequential Implementation

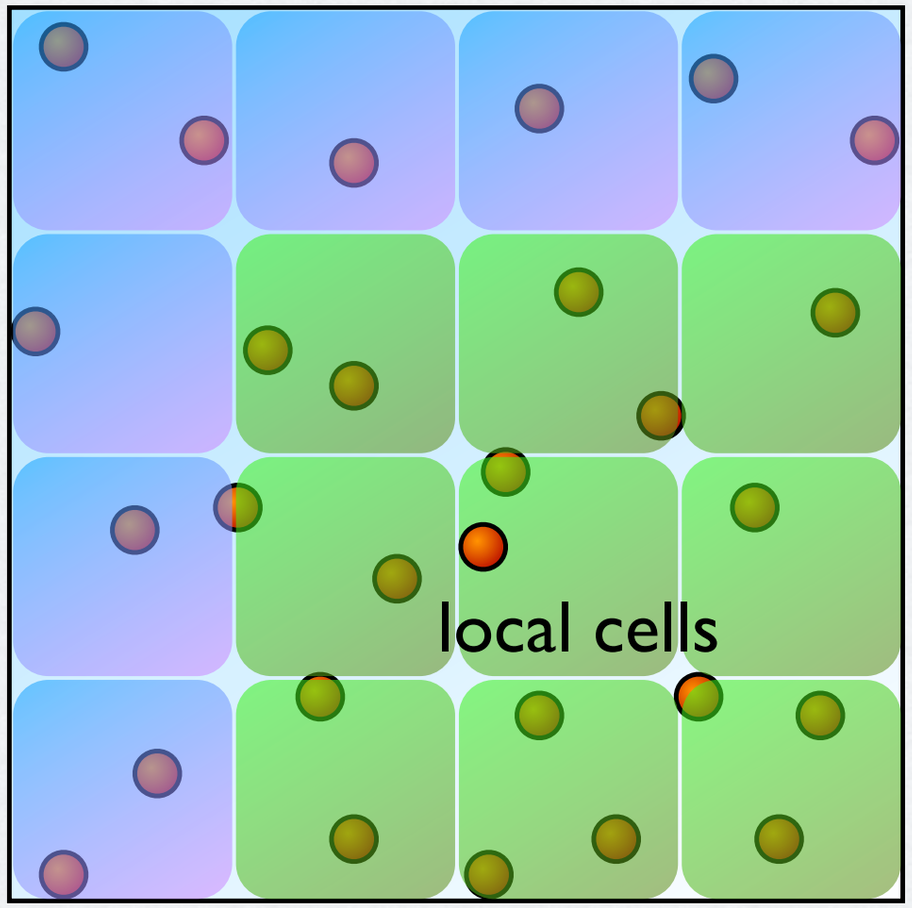
1. For each particle find the ID of its container cell



Sequential Implementation

1. For each particle find the ID of its container cell

2. Form list of particle IDs and corresponding cell IDs

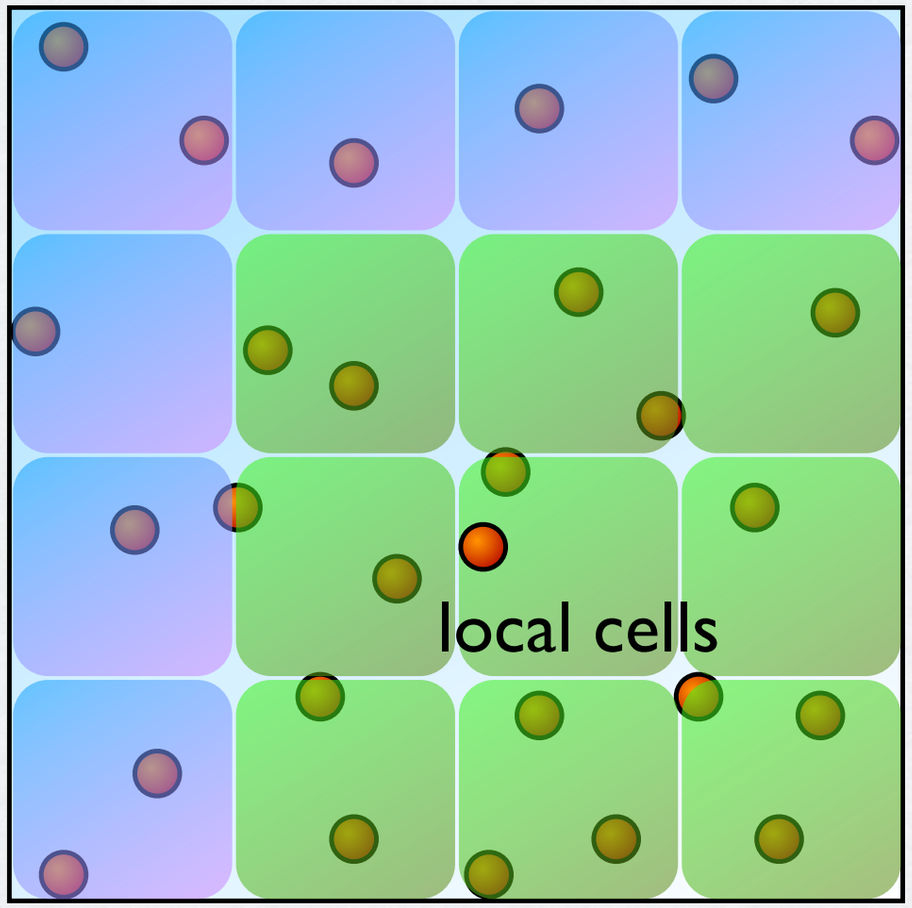


Sequential Implementation

1. For each particle find the ID of its container cell

2. Form list of particle IDs and corresponding cell IDs

3. Sort the particle list by the cell ID



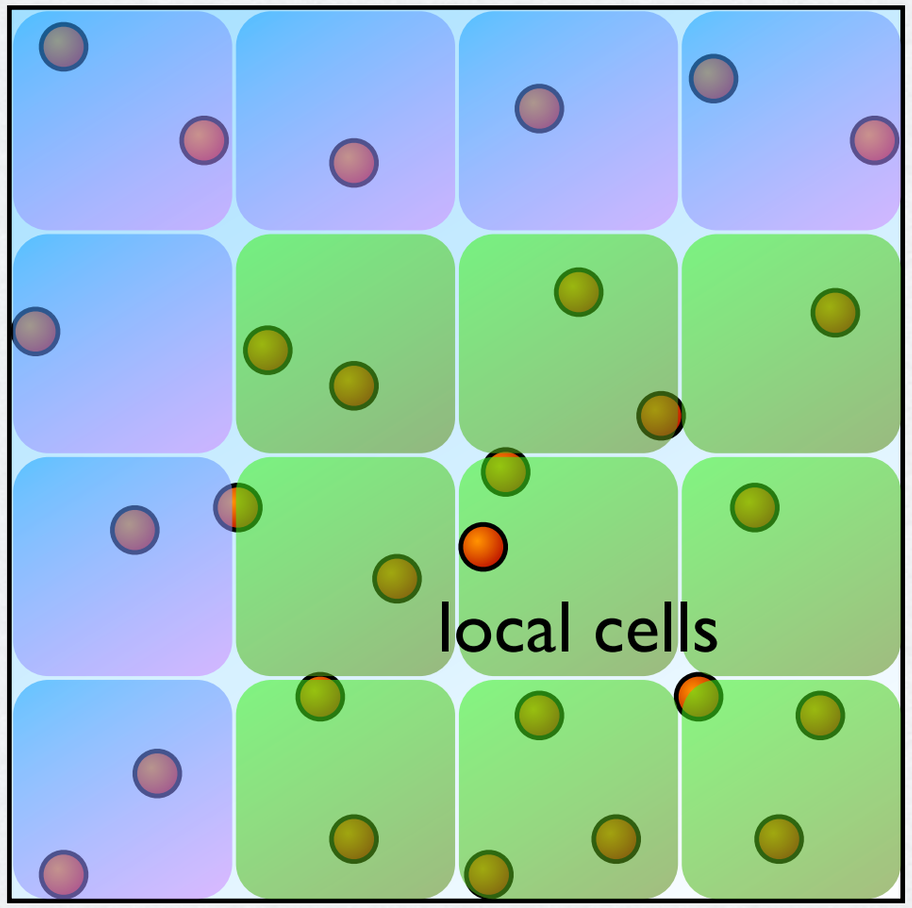
Sequential Implementation

1. For each particle find the ID of its container cell

2. Form list of particle IDs and corresponding cell IDs

3. Sort the particle list by the cell ID

4. Process the cell list according to the cell IDs



Sequential Implementation

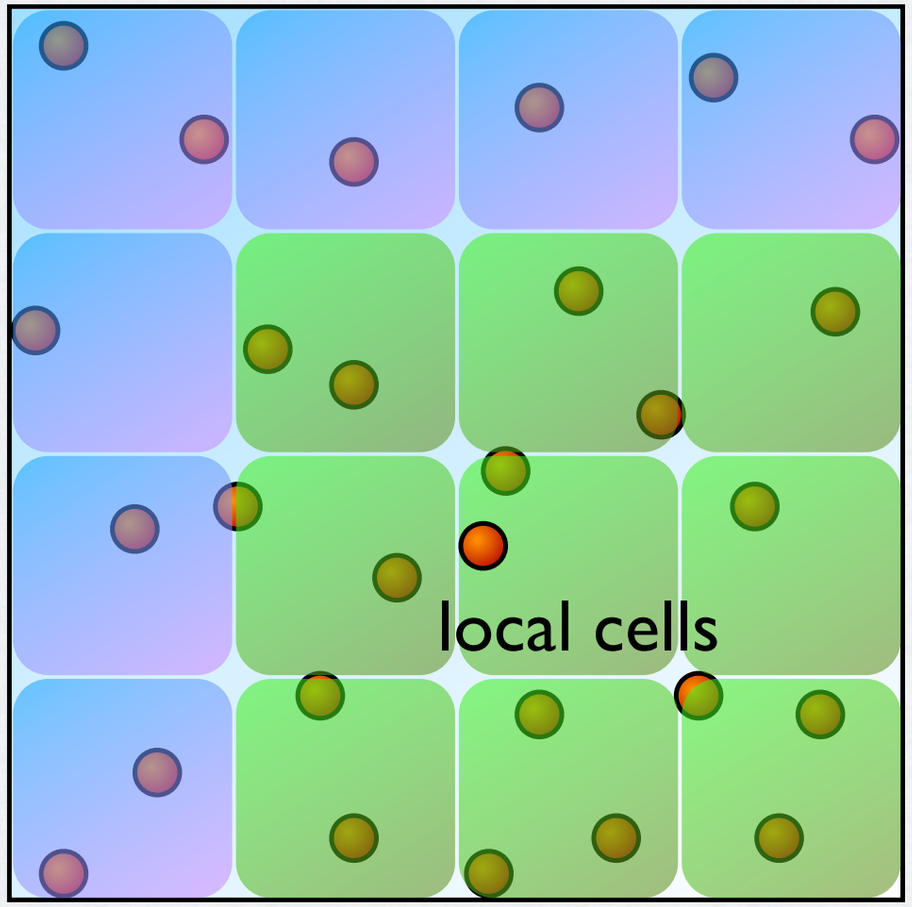
1. For each particle find the ID of its container cell

2. Form list of particle IDs and corresponding cell IDs

3. Sort the particle list by the cell ID

4. Process the cell list according to the cell IDs

5. Perform interactions between all particles with the same cell ID and corresponding neighboring cells



Sequential Implementation

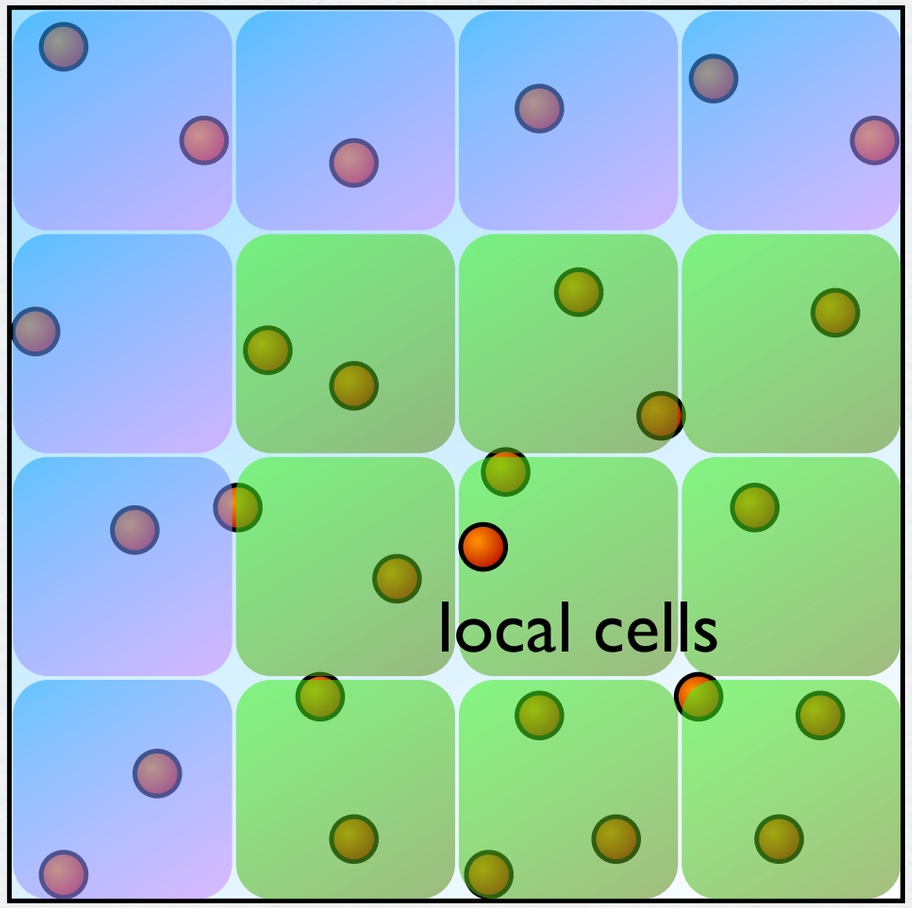
1. For each particle find the ID of its container cell

2. Form list of particle IDs and corresponding cell IDs

3. Sort the particle list by the cell ID

4. Process the cell list according to the cell IDs

5. Perform interactions between all particles with the same cell ID and corresponding neighboring cells

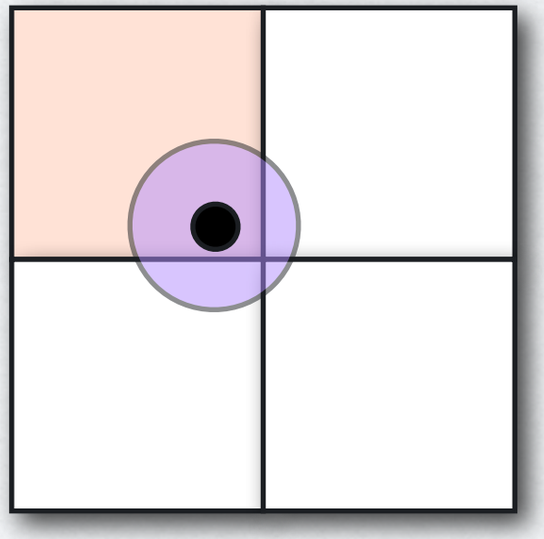


➡ **How to parallelize ?**

Data Structures

For each particle

particle ID
container cell ID



particle-ID	0	1	2	3	4	5	6	7	8	9	10
cell-ID	6	5	8	6	3	4	4	9	7	10	1

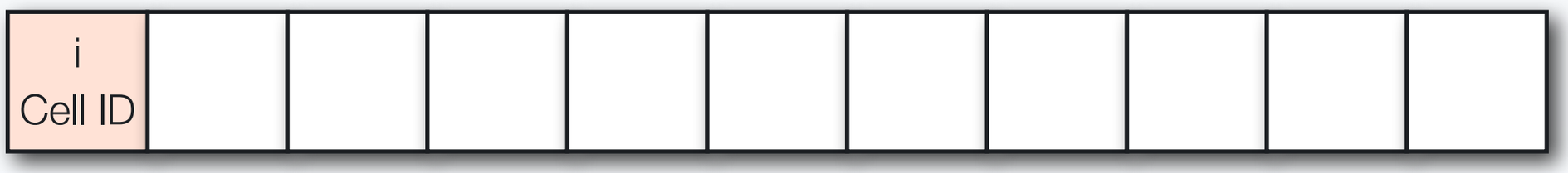
The array contains pairs of (particle-ID, cell-ID)

Constructing the 1D Cell-ID Array

Container cell ID H (hash) for particle i
with position (x_i,y_i,z_i) in]-1,1[^{3}

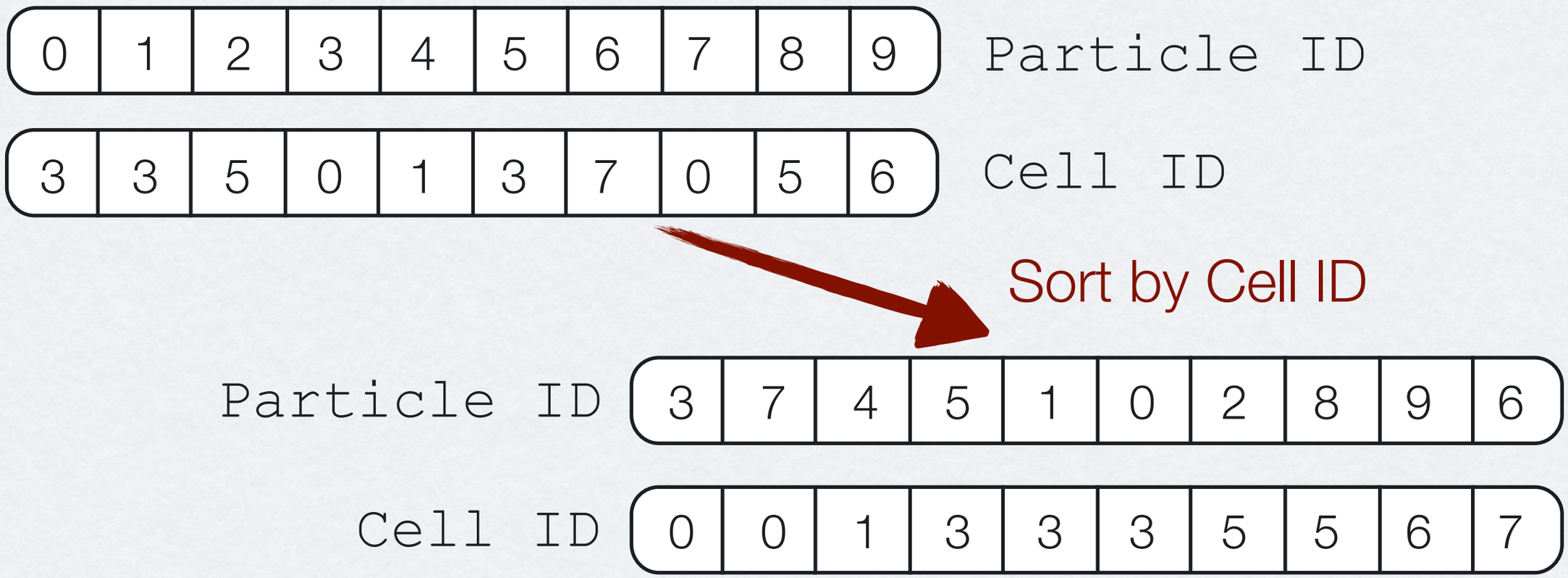
$$\text{cell ID} = \left\lfloor M \frac{x_i + 1}{2} \right\rfloor + M \left\lfloor M \frac{y_i + 1}{2} \right\rfloor + M^2 \left\lfloor M \frac{z_i + 1}{2} \right\rfloor$$

M: number of cells in a given direction



Sorting the Cell ID Array

- Cell ID array is sorted by cell ID
- Particle ID array is also sorted by cell ID
- Cell IDs may appear multiple times
- Radix sort: $\mathcal{O}$(N) sorting



Sorting with C++ - std::sort()

simple:

```
#include <algorithm>
#include <utility>
#include <vector>


//  Custom comparator for pair,
// compares only by the first field
struct compare_first
{
    template <typename T>
    bool operator() (const T& a, const T& b)
    {
     return a.first < b.first;
    }
};

//  Vector storing cellids as first member of pair
// and particleids as second member
std::vector<std::pair<unsigned int, unsigned int> > cellid_particleid(n_particles);


//  Fill the vector with cellids
// x, y, z - arrays storing coordinates of particles
// get_cell_index() returns the index of enclosing cell
for (int i=0; i<n_particles; i++)
    cellid_particleid[i] = std::make_pair(get_cell_index(x[i], y[i], z[i]), i);

std::sort(cellid_particleid.begin(), cellid_particleid.end(), compare_first());
```

BUT: this sorting is $\mathcal{O}$(N logN)!

Creating the Interaction Cell List

Creating the Interaction Cell List

- ***Interaction cell*** *(non-empty cell)*: contains 1 or more particles

Creating the Interaction Cell List

- ***Interaction cell*** *(non-empty cell)*: contains 1 or more particles
- ***Interaction cell list***:
  - created from the sorted cell ID array
  - the sorted cell ID array is scanned for **changes of cell ID**
  - **changes** mark the end of one cell and the beginning of another.

Creating the Interaction Cell List

- ***Interaction cell*** *(non-empty cell)*: contains 1 or more particles
- ***Interaction cell list***:
  - created from the sorted cell ID array
  - the sorted cell ID array is scanned for **changes of cell ID**
  - **changes** mark the end of one cell and the beginning of another.
- ***Parallelization***: each thread is assigned a (roughly equal) segment of the cell ID array to scan for changes

Creating the Interaction Cell List

- ***Parallelization***: threads assigned to portions of the 1D array.

Creating the Interaction Cell List

- ***Parallelization***: threads assigned to portions of the 1D array.
- *HOW TO*:
  - each thread scans past the end point of its pre-assigned segment and stops when it finds a change in cell-ID (except for the last thread)
  - each thread skips its first change as it has been found by a thread assigned to the preceding segment(except for the first thread)

Creating the Interaction Cell List

- The sorted 1D cell ID array is scanned **twice**

Creating the Interaction Cell List

- The sorted 1D cell ID array is scanned **twice**

  - `First scan`
    - count the number of particles within each interaction cell
    - convert them into offsets in the 1D array (through a parallel prefix sum in several passes)

Creating the Interaction Cell List

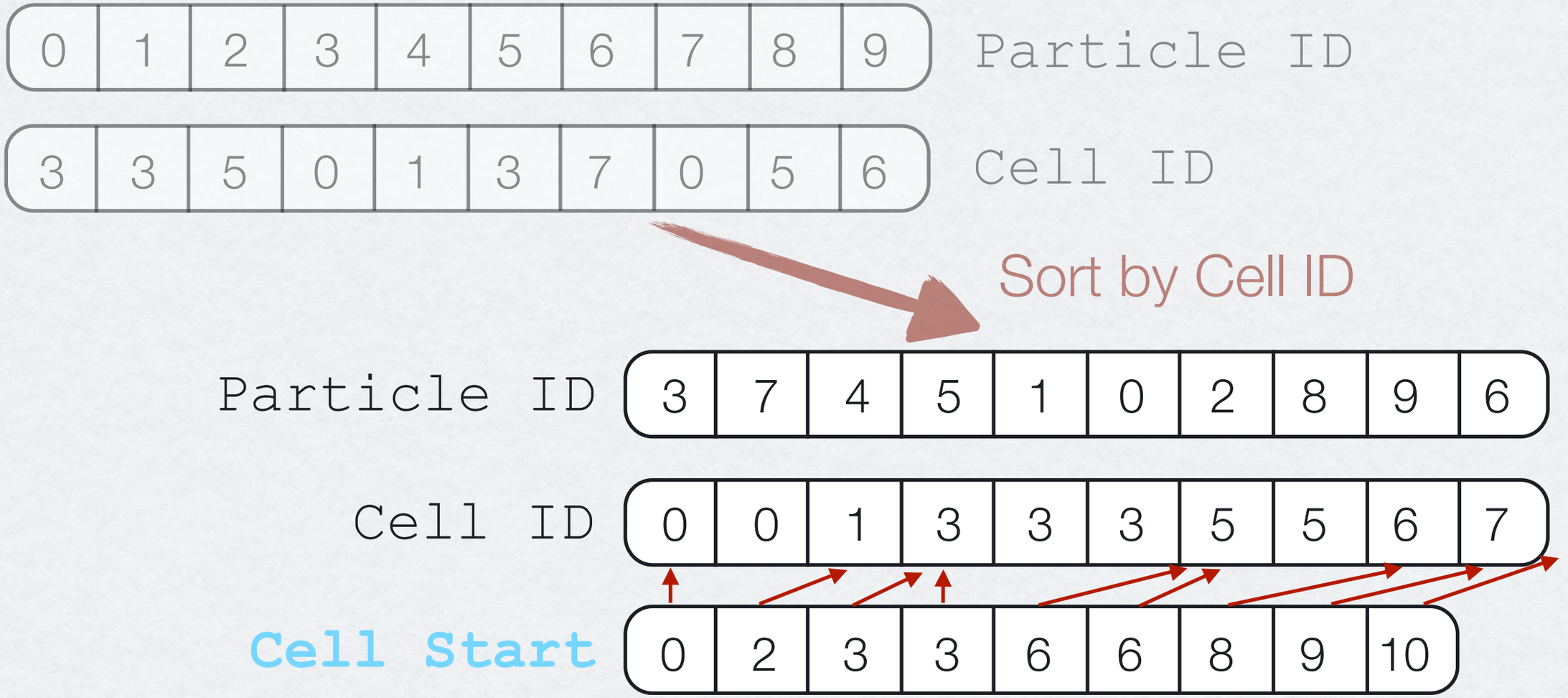
- The sorted 1D cell ID array is scanned **twice**

  - First scan
    - count the number of particles within each interaction cell
    - convert them into offsets in the 1D array (through a parallel prefix sum in several passes)

  - Second scan
    - create entries for each interaction cell in a **new array (“Cell Start”):**
    - **This array tells at which index in the 1D cell-ID array starts each cell**

Sorting the Cell ID Array

- Cell Start array marks the first particle in Particle ID array that is contained by the cell
- The size of Cell Start array is the number of cells plus one
- The last element always marks the end of Particle ID array



Cell Start Array

In C++ Cell Start calculation is straightforward

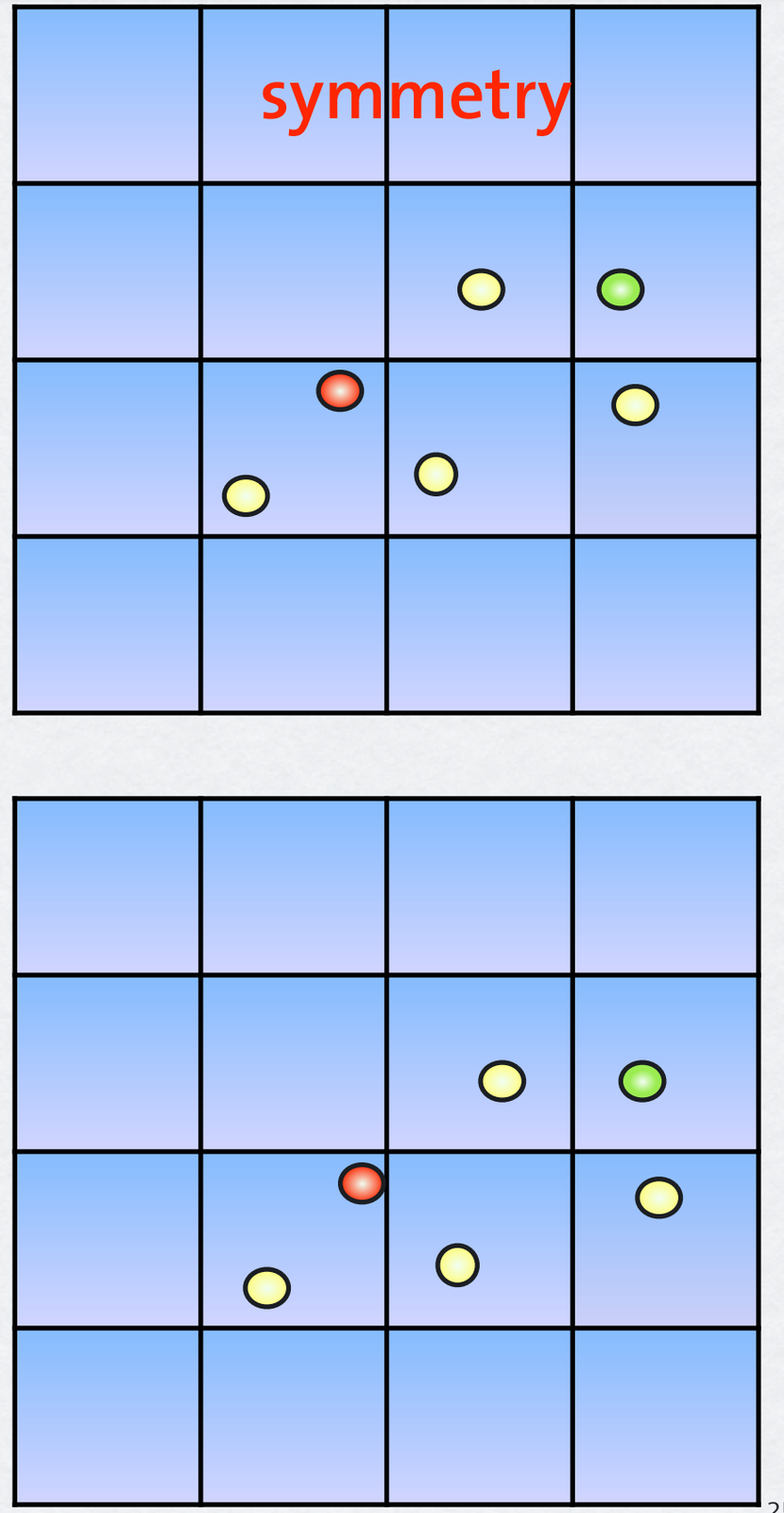
```
int current_cell = -1;
for (int i = 0; i < n_particles; i++)
{
    for (int j = current_cell; j < cellids[i]; j++)
    {
        start[j+1] = i;
    }
    current_cell = cellids[i];
}

for (int j = current_cell; j <= n_cells; j++)
    start[j+1] = n_particles;
```

Traversing the list

- After the creation of the list all the **particle neighbors are found** by:
  - **visiting all the cells that are adjacent to the cell containing the target particle**
- This make up a total of 9/27 (2D/3D) cells to look up
- Additional care has to be taken for the cells on the border of domain.
  Special case : Periodic Boundary Conditions.

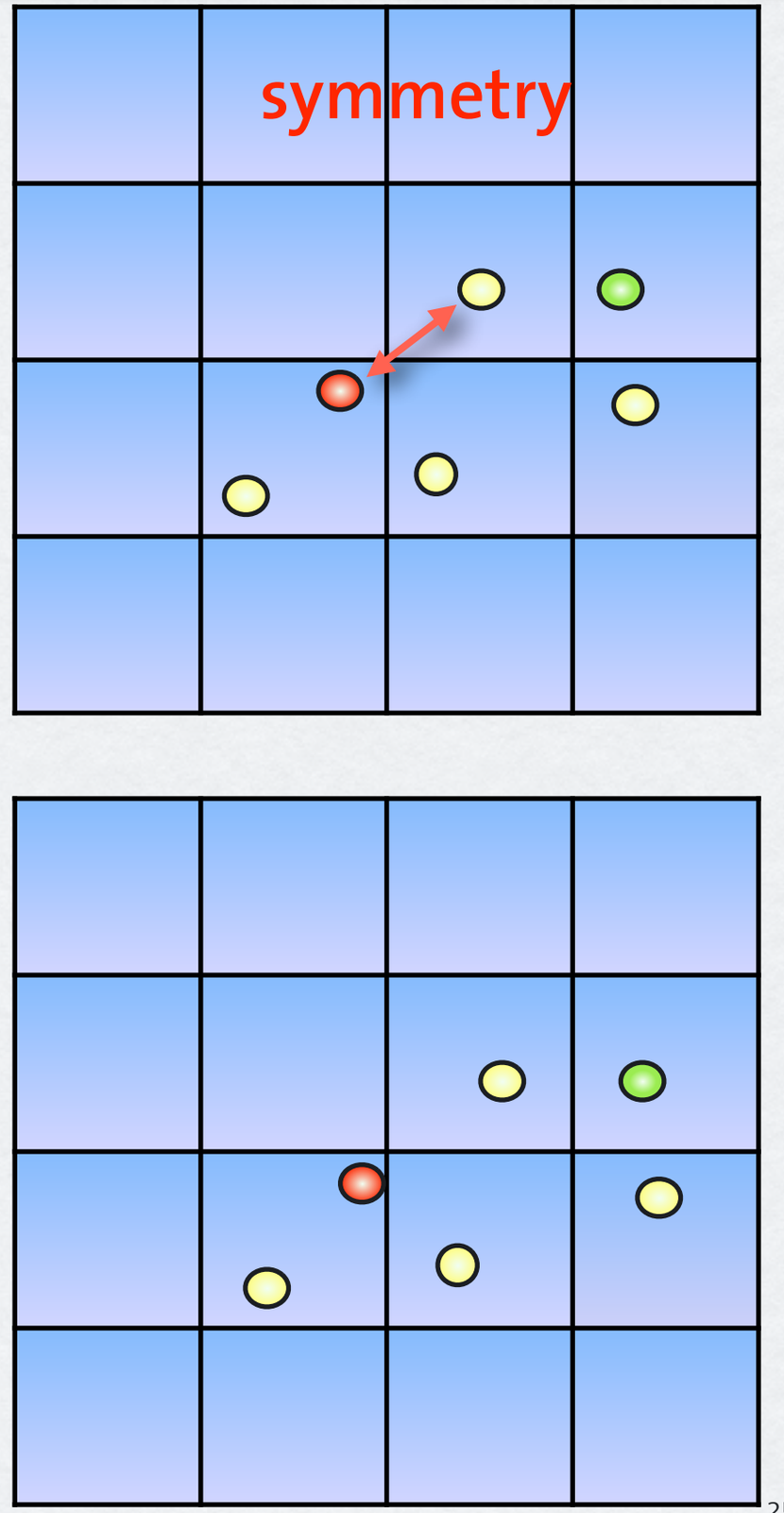
Parallelization over Particles

- one thread processes one particle
  - particle of thread 1
  - particle of thread 2
  - other particles



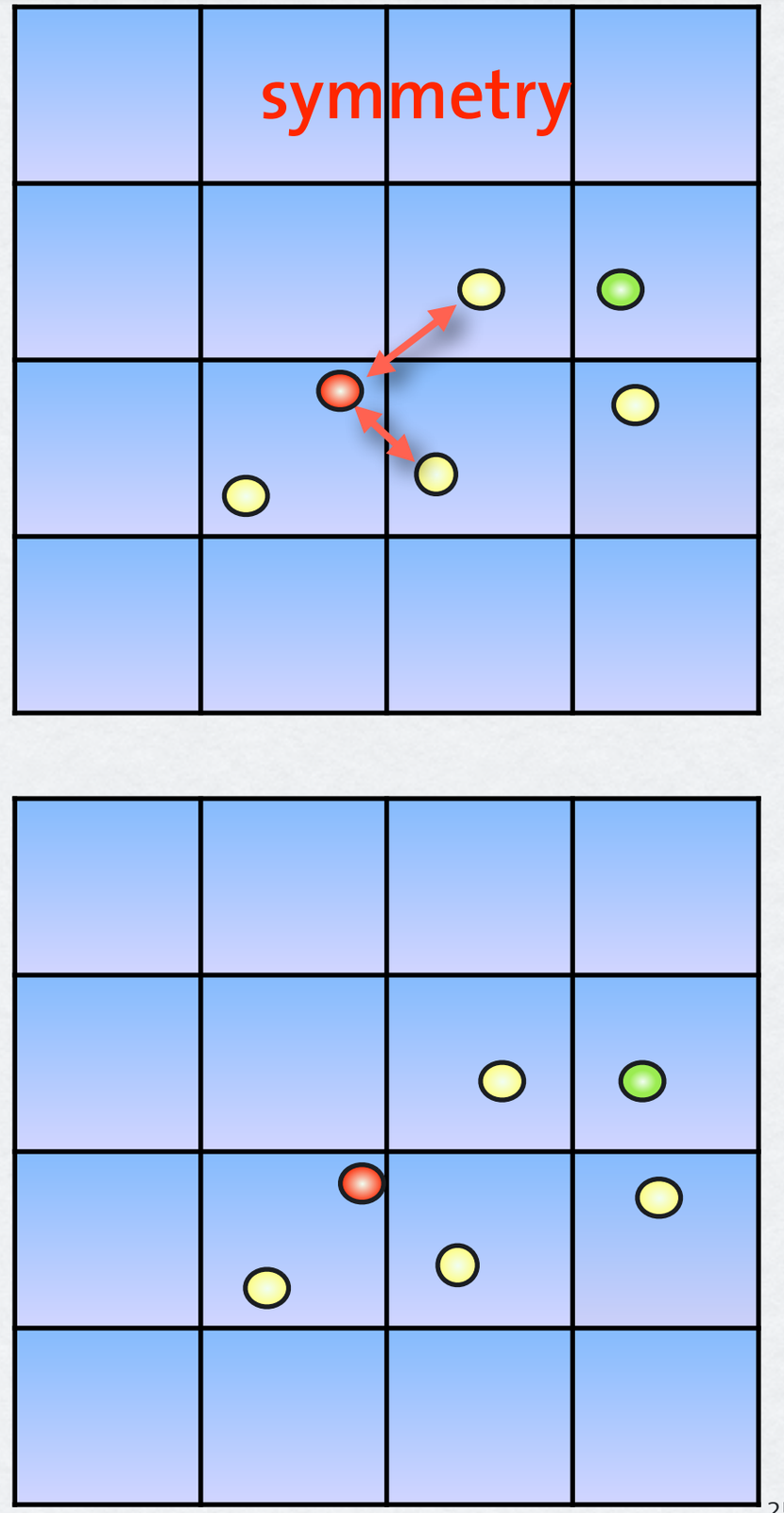
Parallelization over Particles

- one thread processes one particle
  - particle of thread 1
  - particle of thread 2
  - other particles



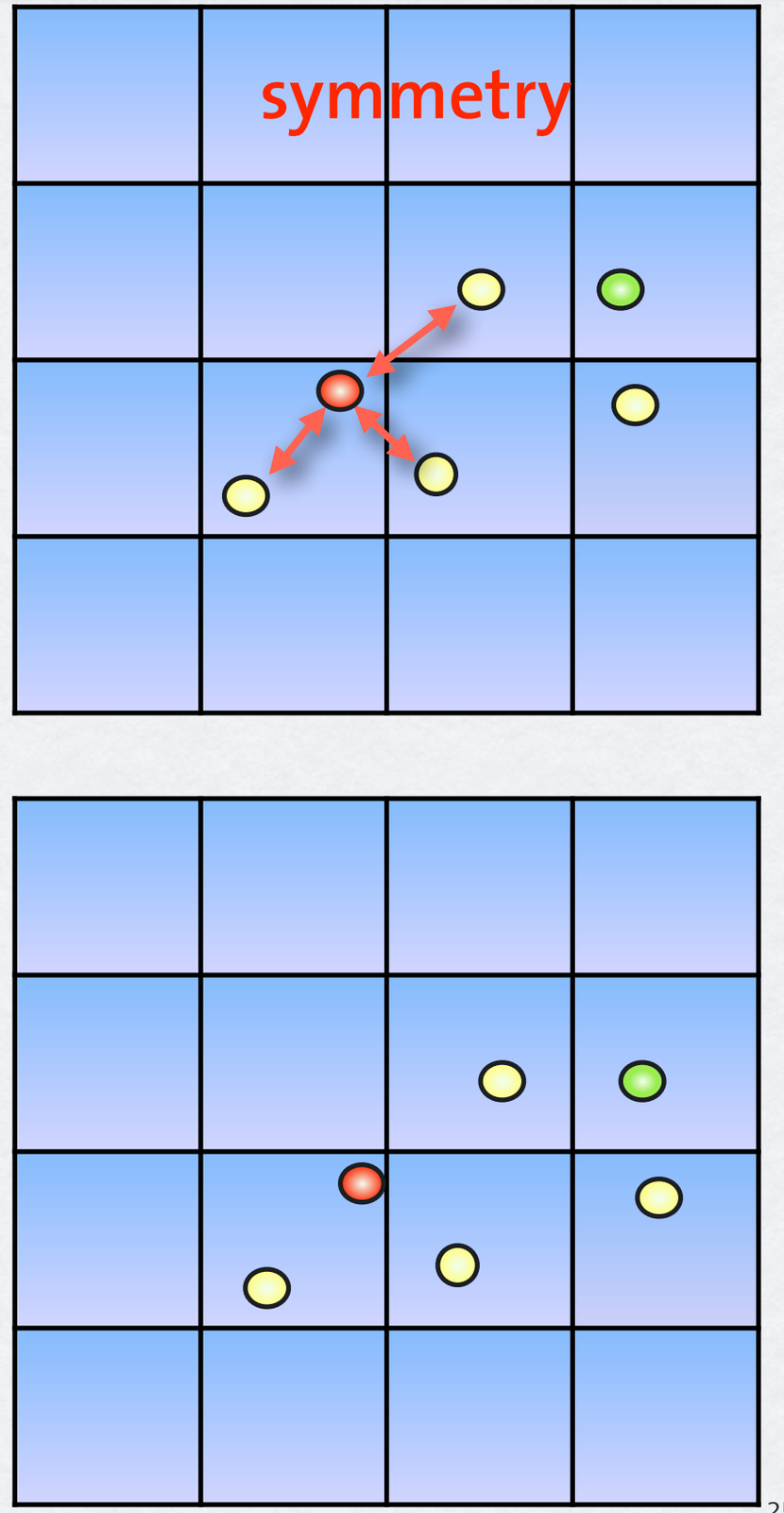
Parallelization over Particles

- one thread processes one particle
  - particle of thread 1
  - particle of thread 2
  - other particles



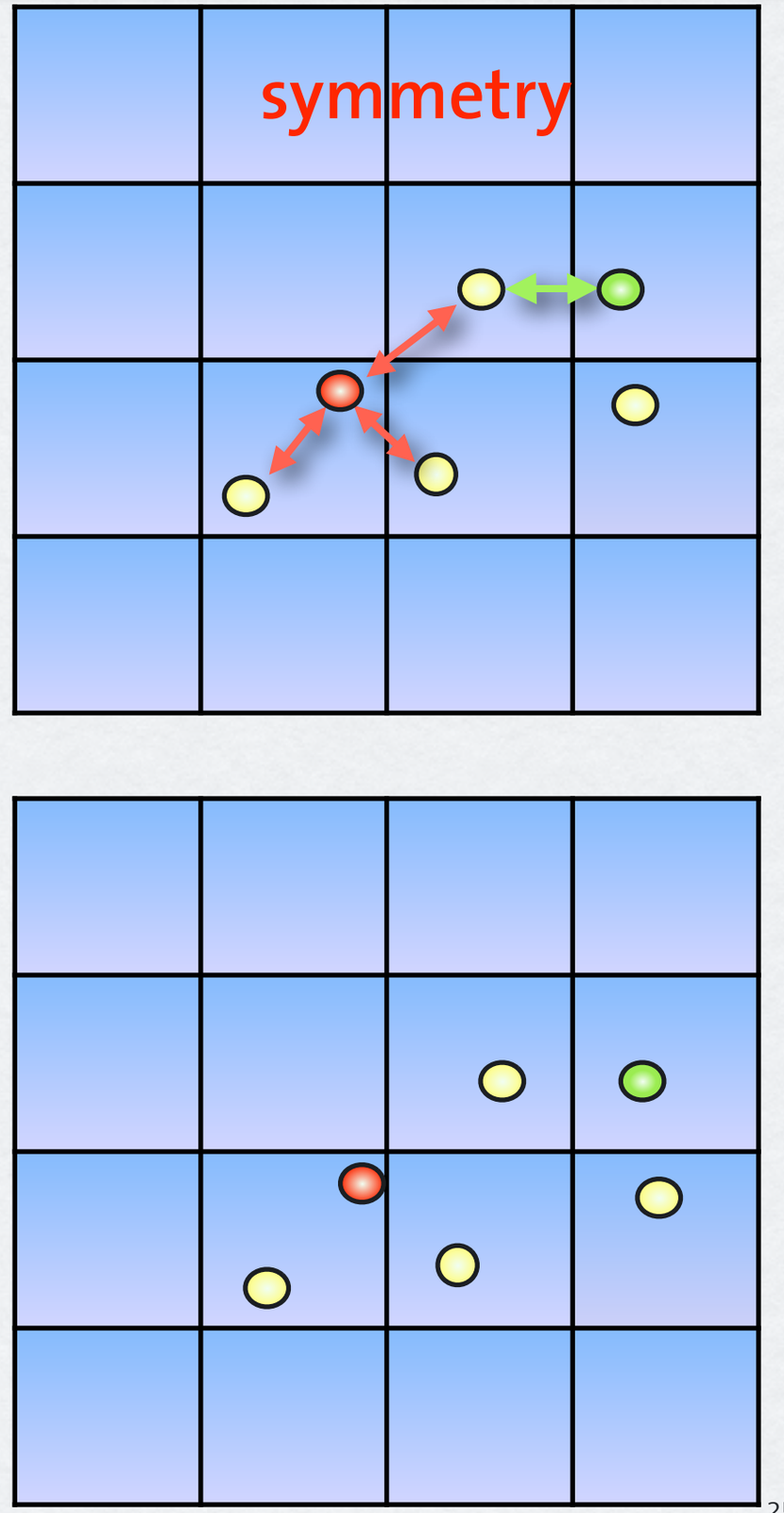
Parallelization over Particles

- one thread processes one particle
  - particle of thread 1
  - particle of thread 2
  - other particles



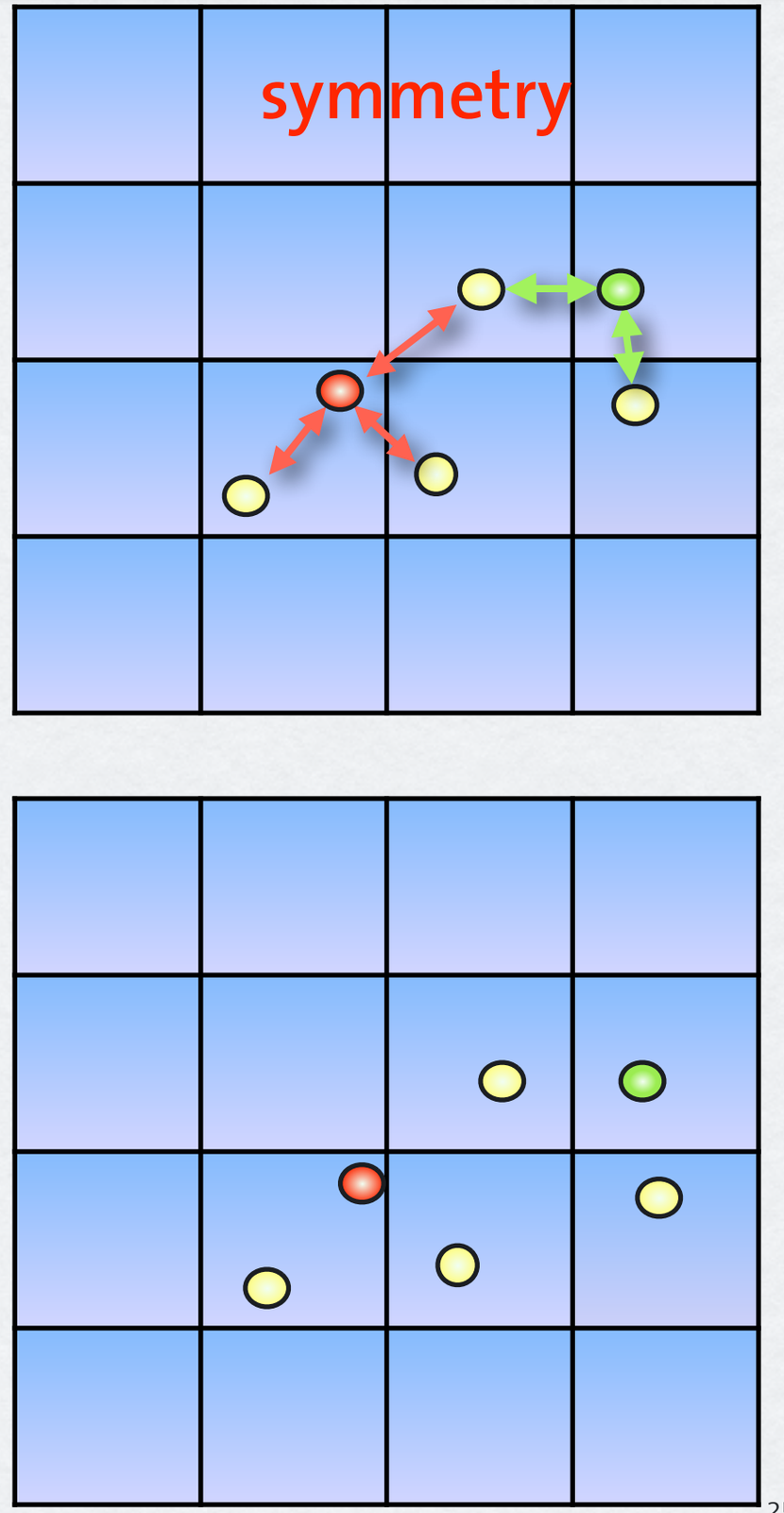
Parallelization over Particles

- one thread processes one particle
  - particle of thread 1
  - particle of thread 2
  - other particles



Parallelization over Particles

- one thread processes one particle
  - particle of thread 1
  - particle of thread 2
  - other particles



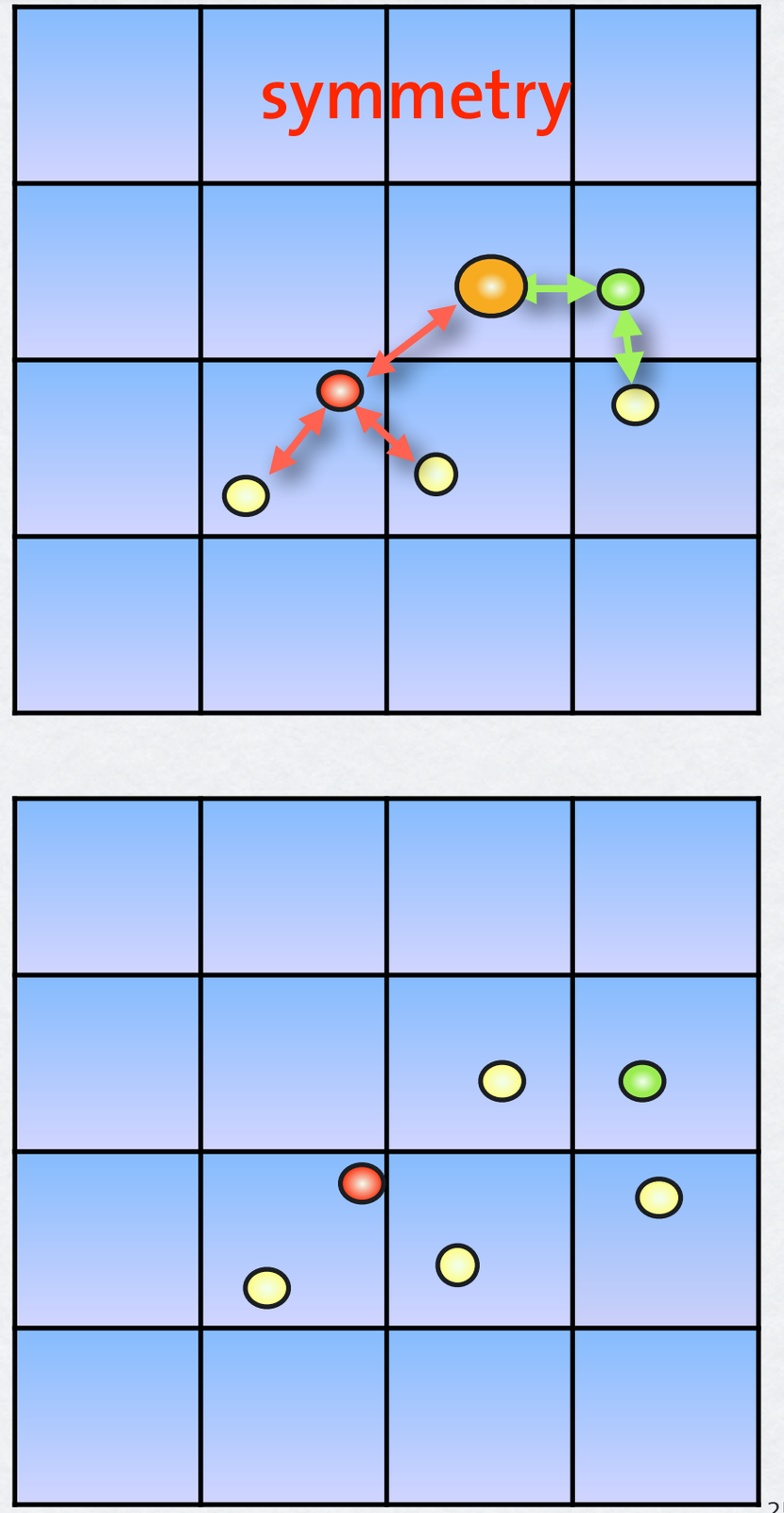
Parallelization over Particles

➡ one thread processes one particle

- particle of thread 1
- particle of thread 2
- other particles

PROBLEMS

- race condition!



Parallelization over Particles

➡ one thread processes one particle

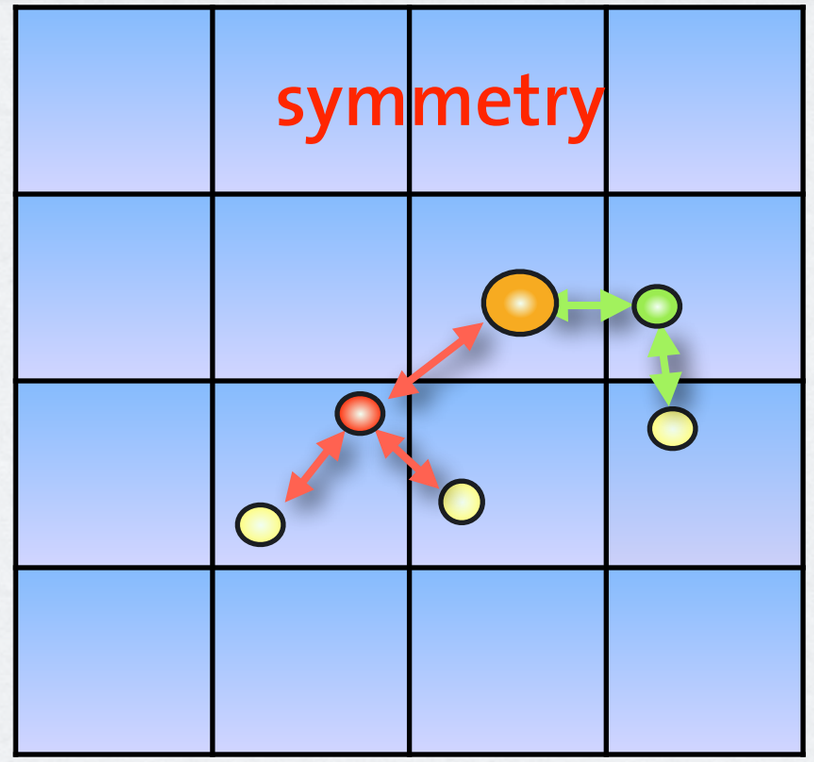
- particle of thread 1
- particle of thread 2
- other particles

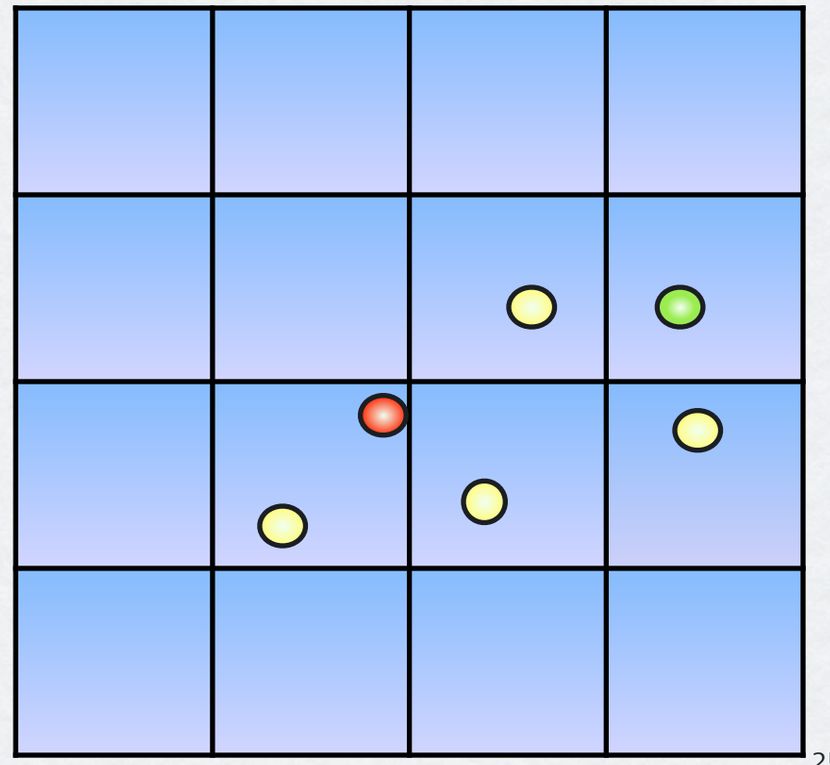
PROBLEMS

- race condition!

- load imbalance

(some threads do more work than others)

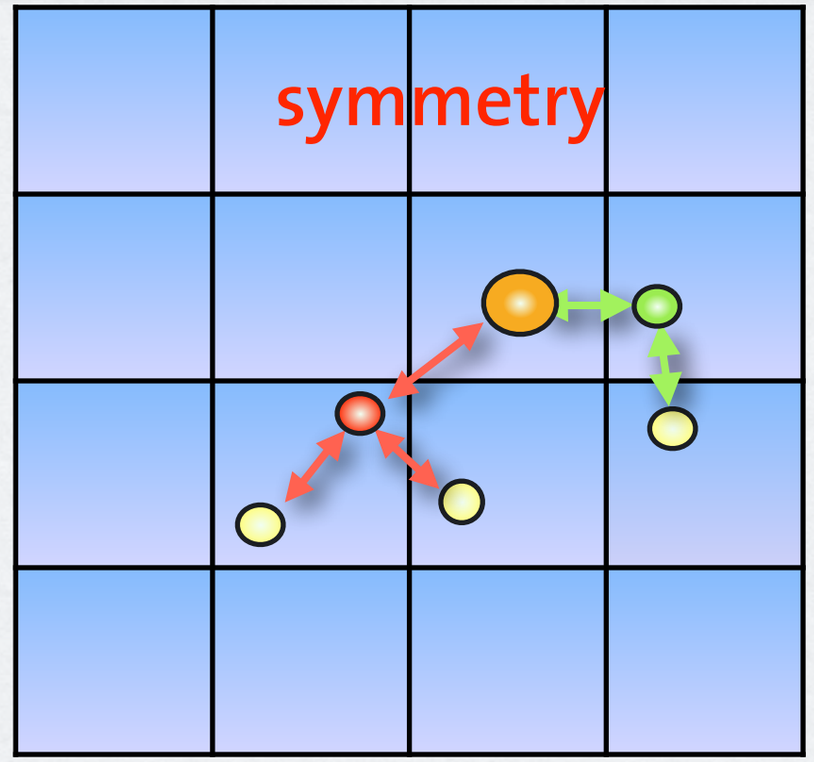




Parallelization over Particles

➡ one thread processes one particle

- particle of thread 1
- particle of thread 2
- other particles

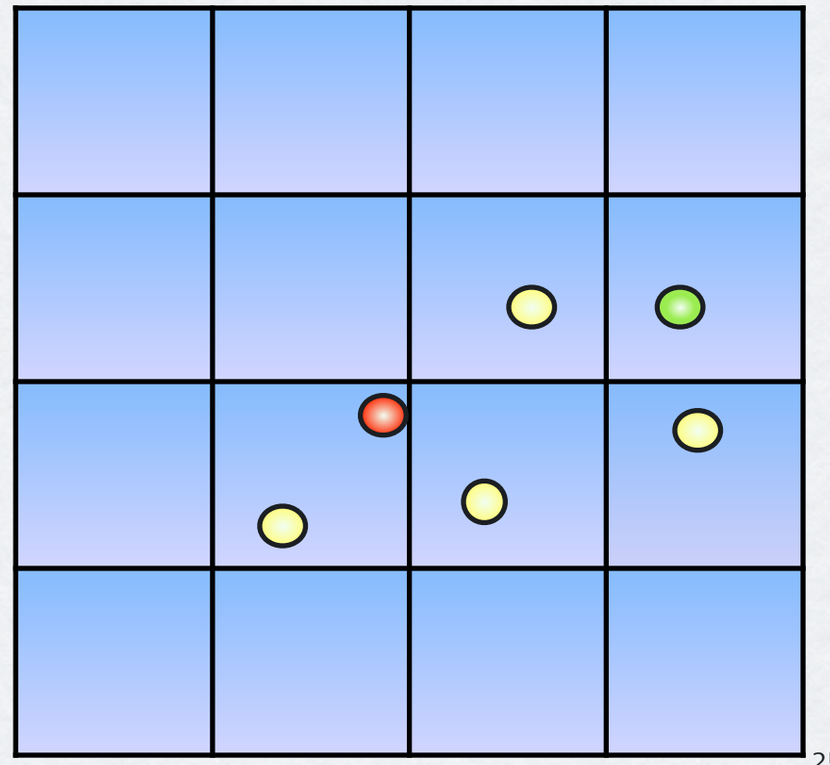


PROBLEMS

- race condition!

- load imbalance

(some threads do more work than others)

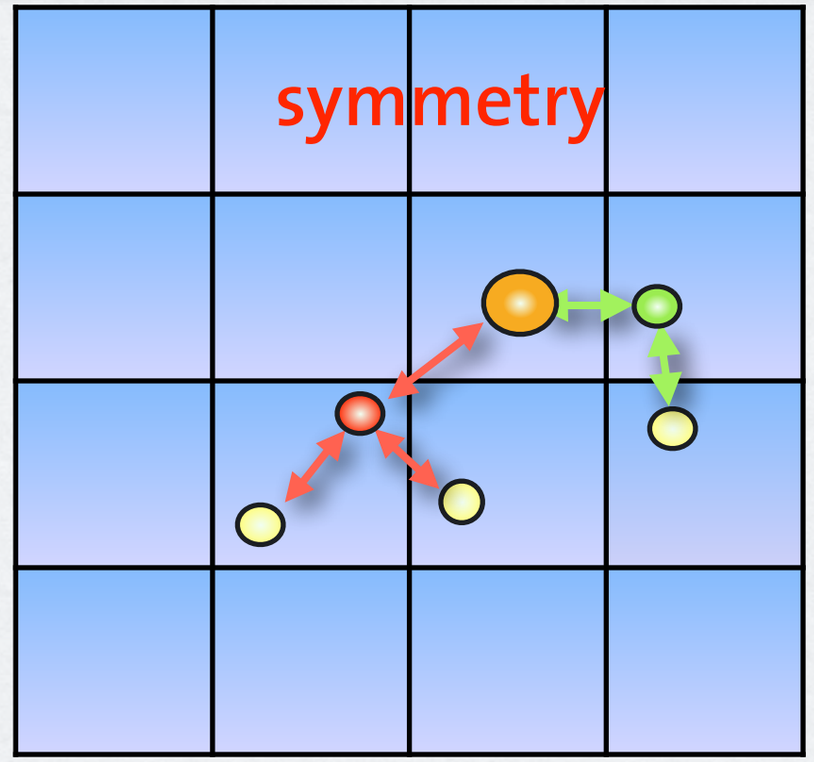


HOW TO FIX IT:

Parallelization over Particles

➡ one thread processes one particle

- particle of thread 1
- particle of thread 2
- other particles

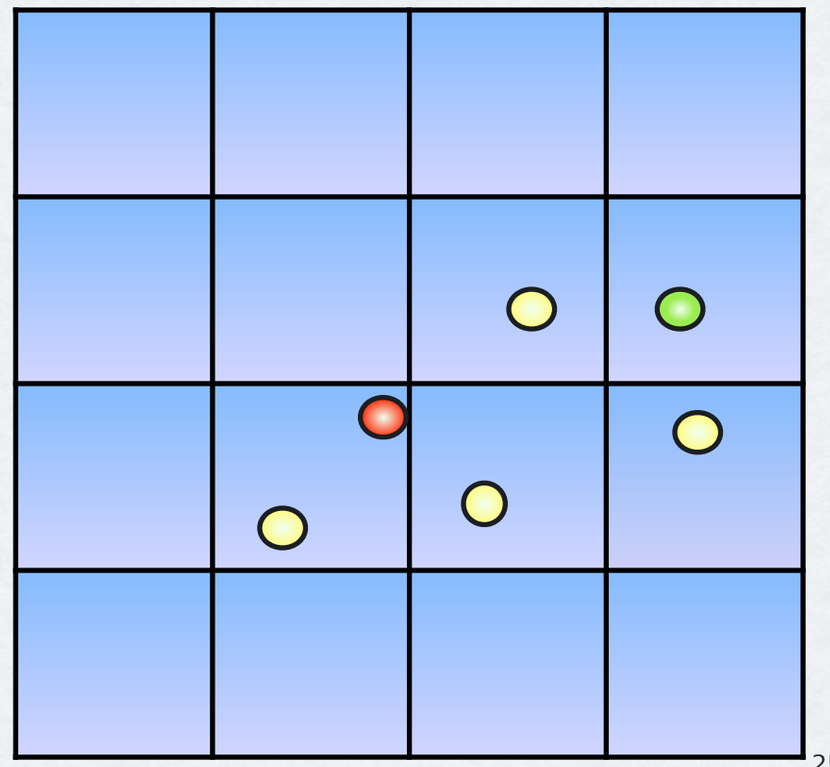


PROBLEMS

- race condition!

- load imbalance

(some threads do more work than others)



HOW TO FIX IT:

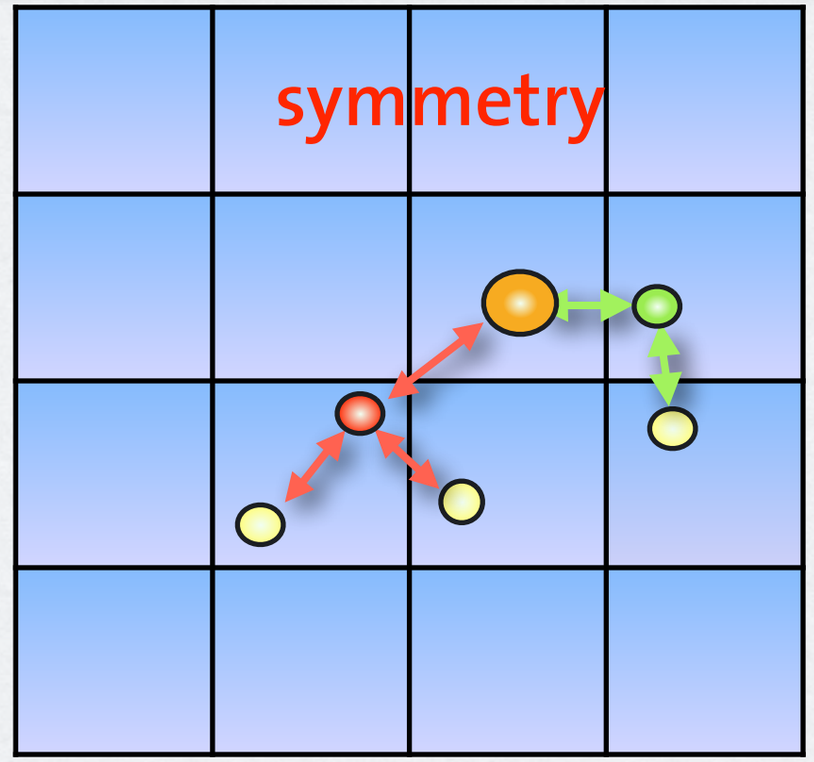
☑ remove symmetry to avoid race conditions : each particle is updated separately

☑ Tasking

Parallelization over Particles

➡ one thread processes one particle

- particle of thread 1
- particle of thread 2
- other particles

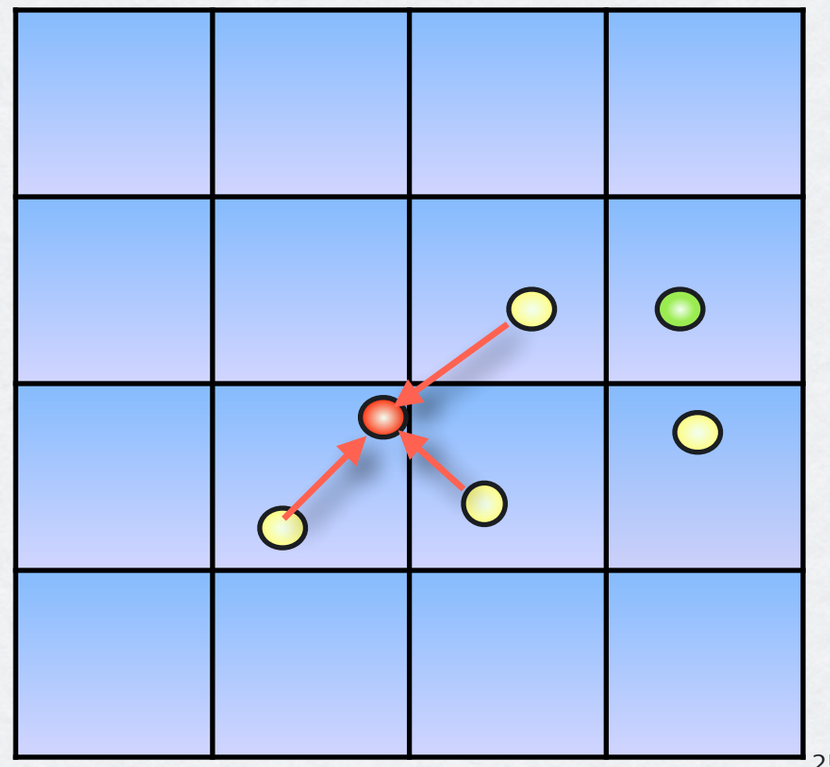


PROBLEMS

- race condition!

- load imbalance

(some threads do more work than others)



HOW TO FIX IT:

☑ remove symmetry to avoid race conditions : each particle is updated separately

☑ Tasking

Parallelization over Particles

➡ one thread processes one particle

- particle of thread 1
- particle of thread 2
- other particles

PROBLEMS

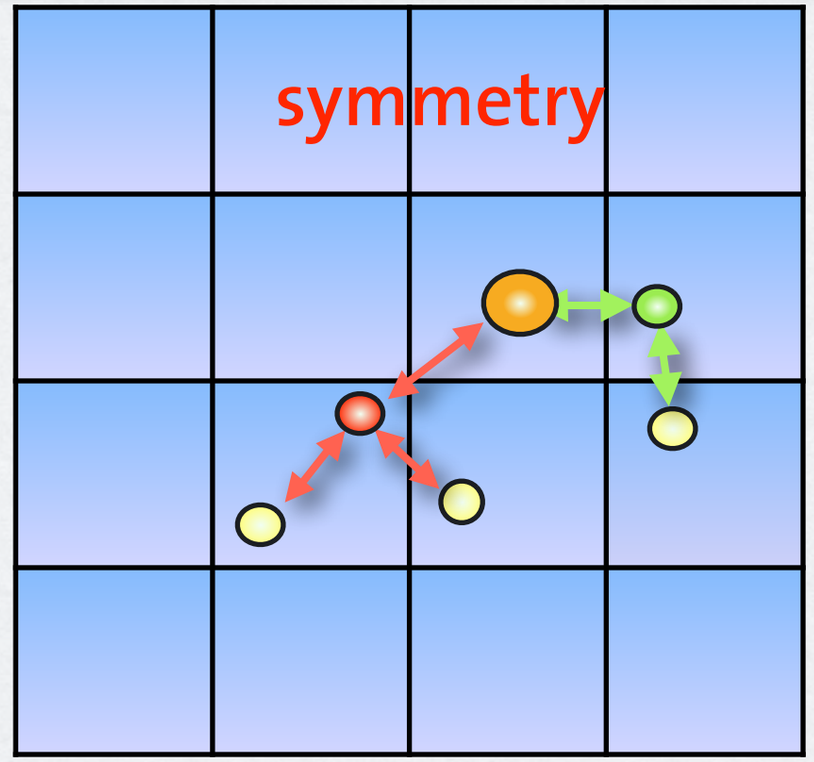
- race condition!

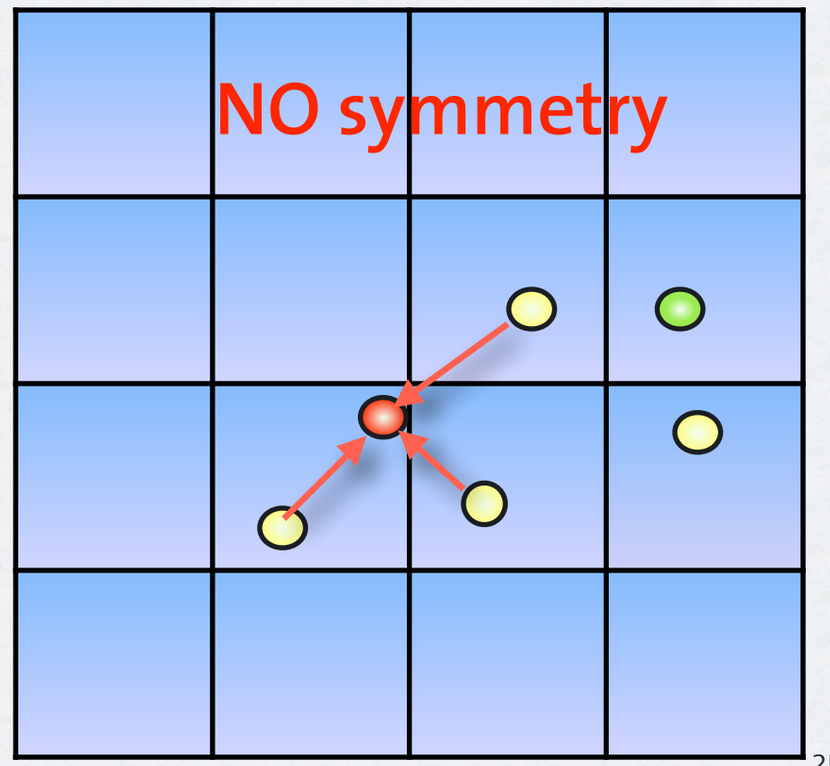
- load imbalance
  (some threads do more work than others)

HOW TO FIX IT:

☑ remove symmetry to avoid race conditions : each particle is updated separately

☑ Tasking





Parallelization over Particles

➡ one thread processes one particle

- particle of thread 1
- particle of thread 2
- other particles

PROBLEMS

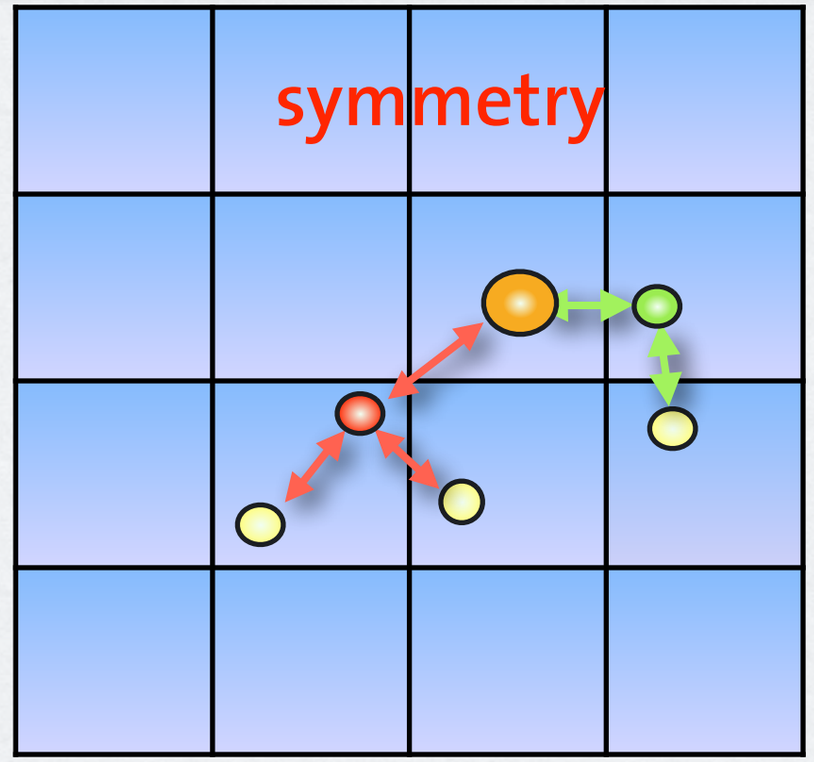
- race condition!

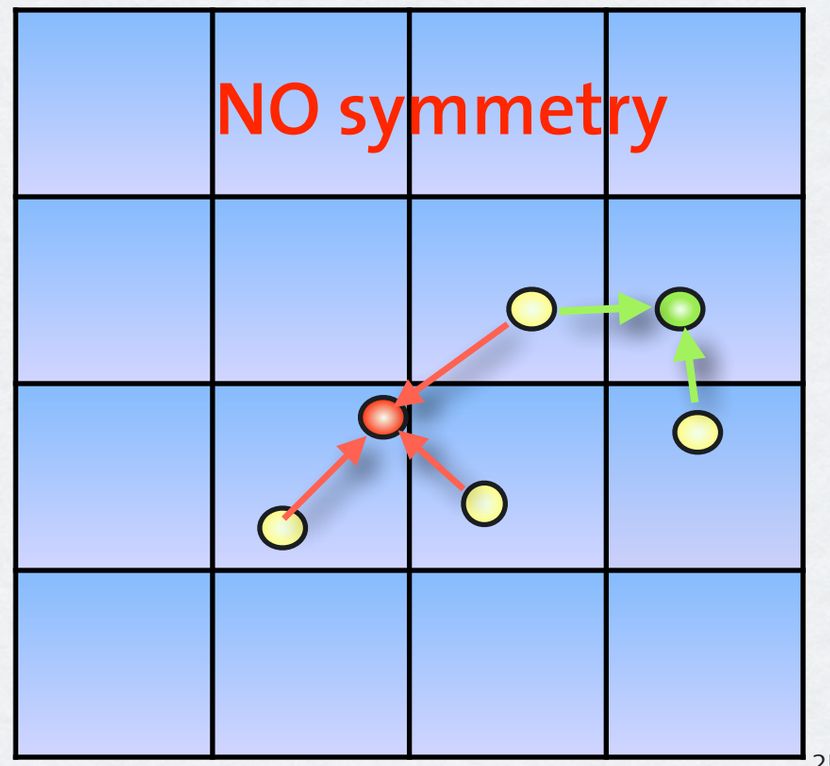
- load imbalance
  (some threads do more work than others)

HOW TO FIX IT:

☑ remove symmetry to avoid race conditions : each particle is updated separately

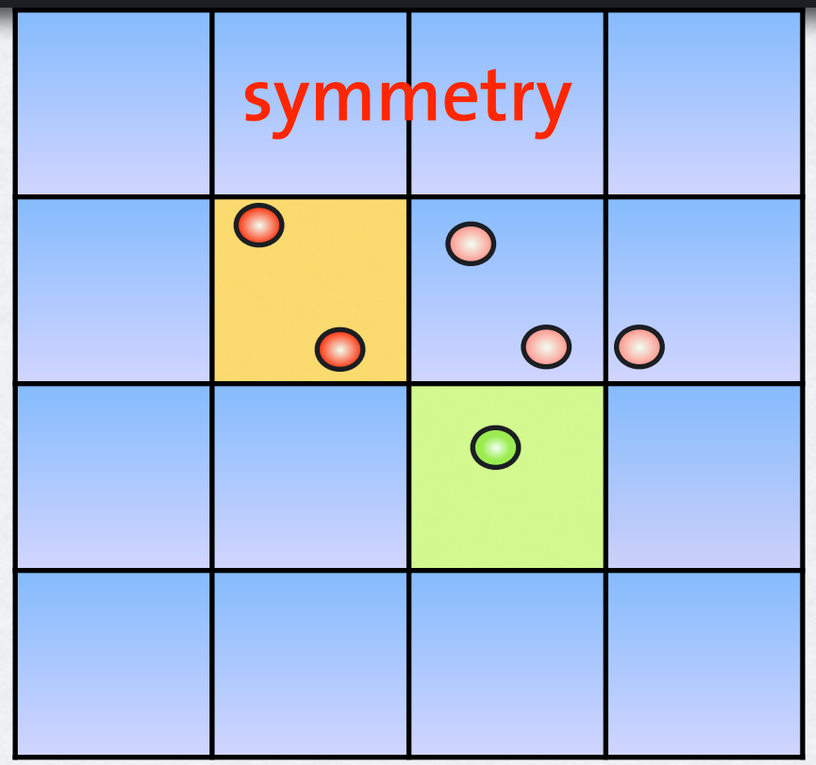
☑ Tasking





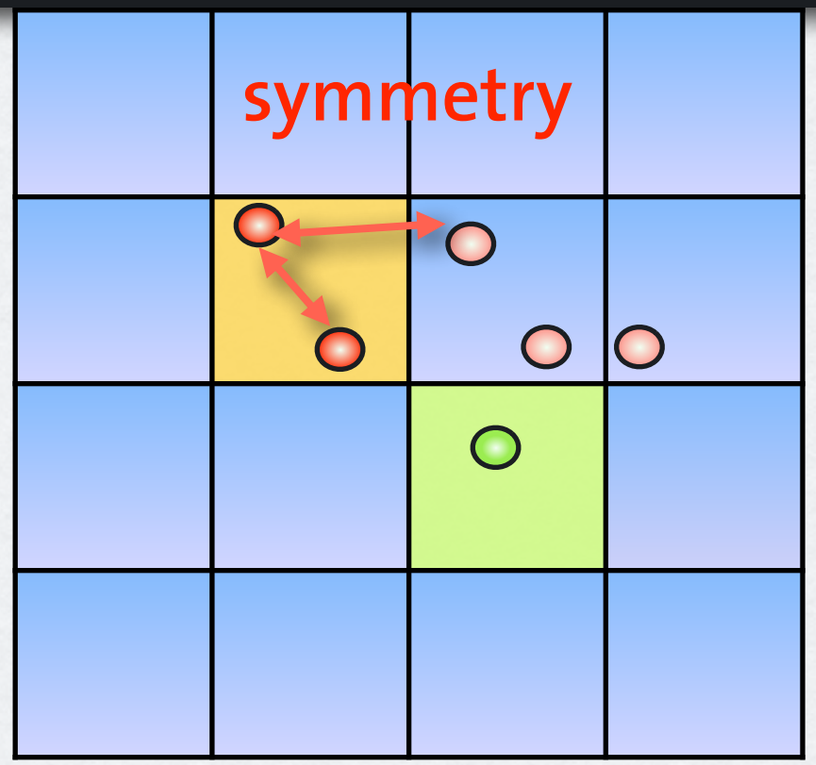
Parallelization over Cells

- ➡ one thread processes one cell
  - particles in cell of thread 1
  - particles in cell of thread 2
  - particles in other cells



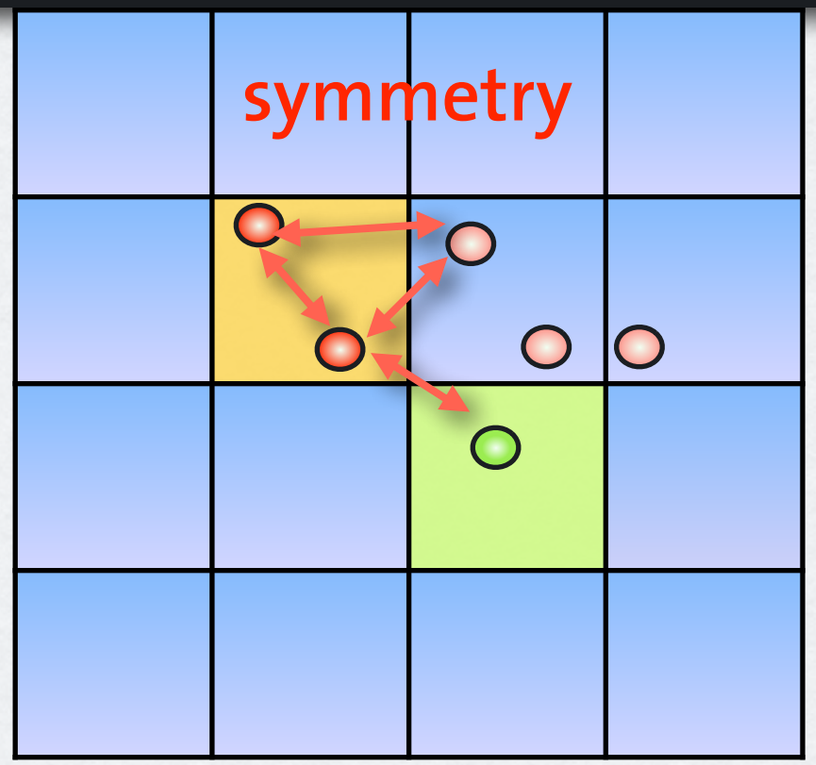
Parallelization over Cells

- ➡ one thread processes one cell
  - particles in cell of thread 1
  - particles in cell of thread 2
  - particles in other cells



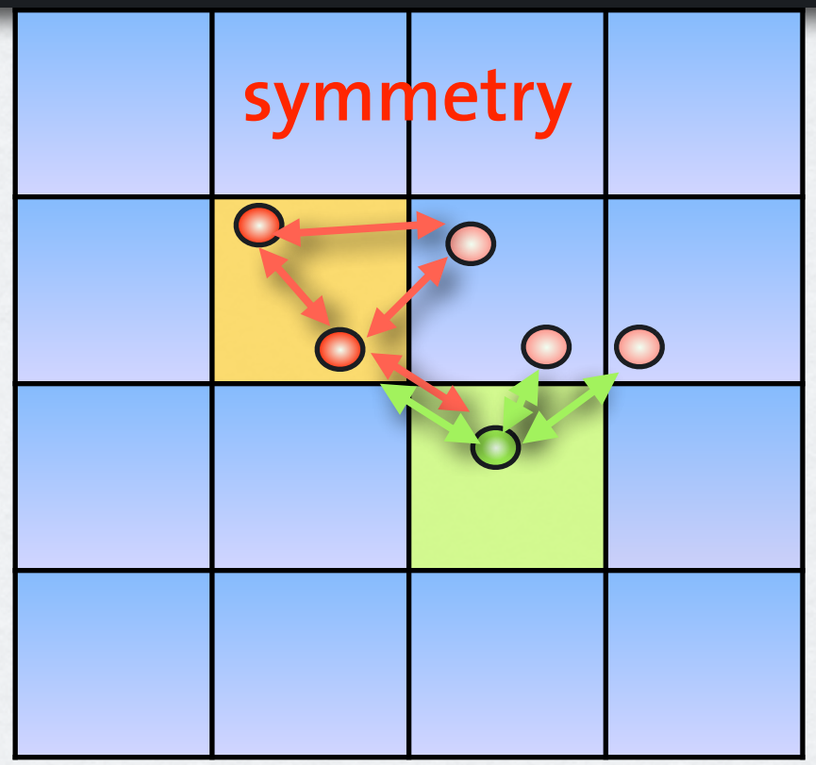
Parallelization over Cells

- ➡ one thread processes one cell
  - particles in cell of thread 1
  - particles in cell of thread 2
  - particles in other cells



Parallelization over Cells

- ➡ one thread processes one cell
  - particles in cell of thread 1
  - particles in cell of thread 2
  - particles in other cells



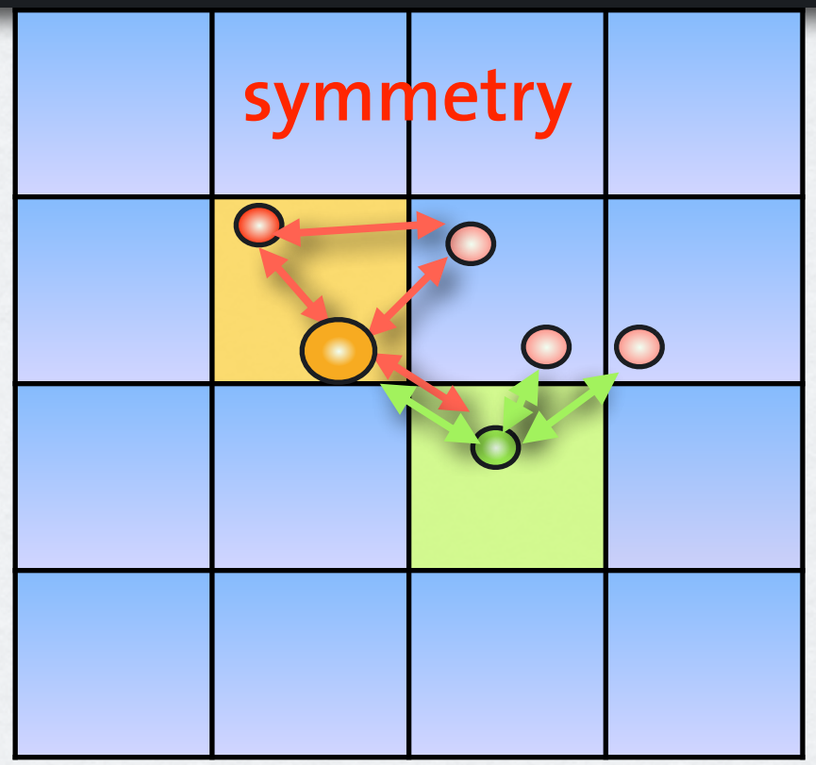
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

- race condition!



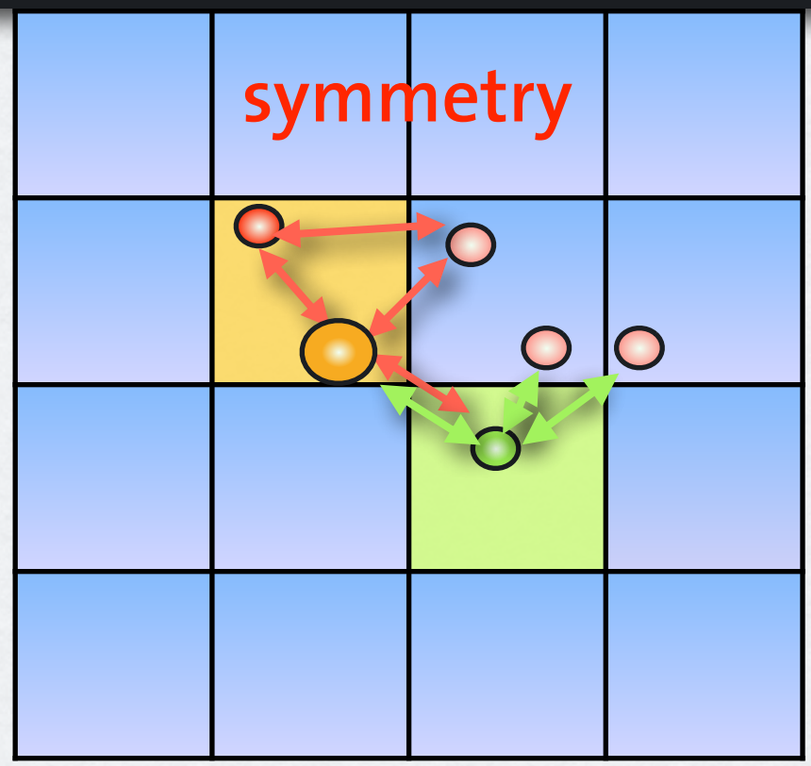
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

- race condition!
- load imbalance (e.g. if some cell is empty)



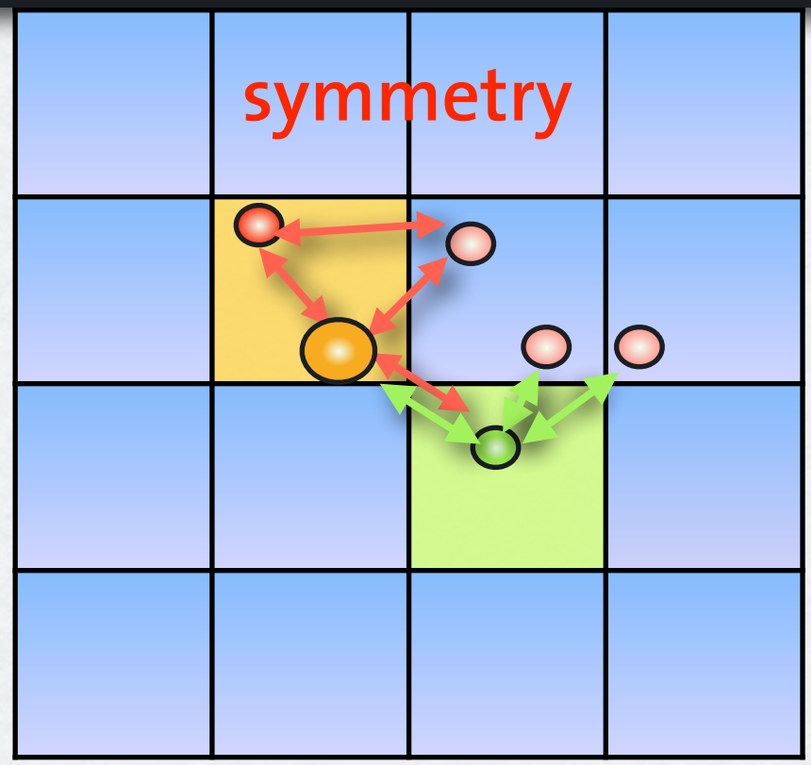
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

- race condition!
- load imbalance (e.g. if some cell is empty)



How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task

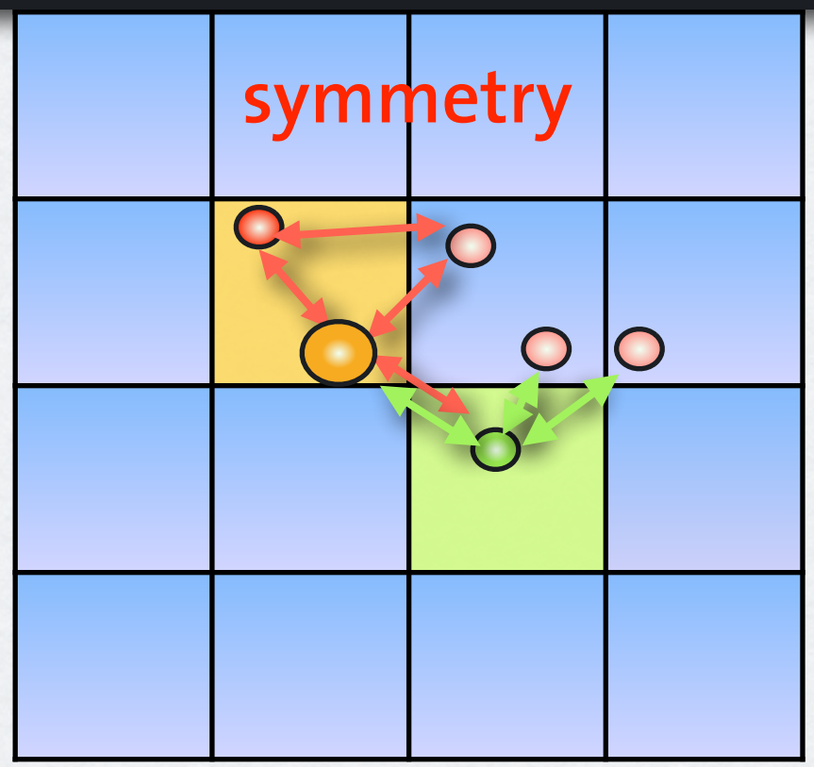
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

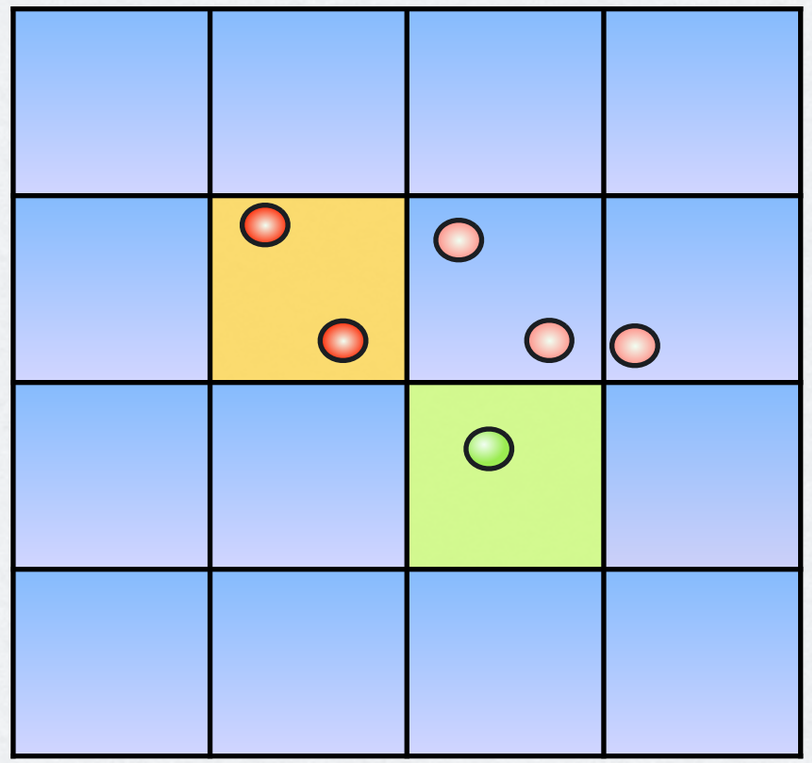
POTENTIAL PROBLEMS

- race condition!
- load imbalance (e.g. if some cell is empty)



How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task



How to fix race condition:

- ☑ each particle updates just itself

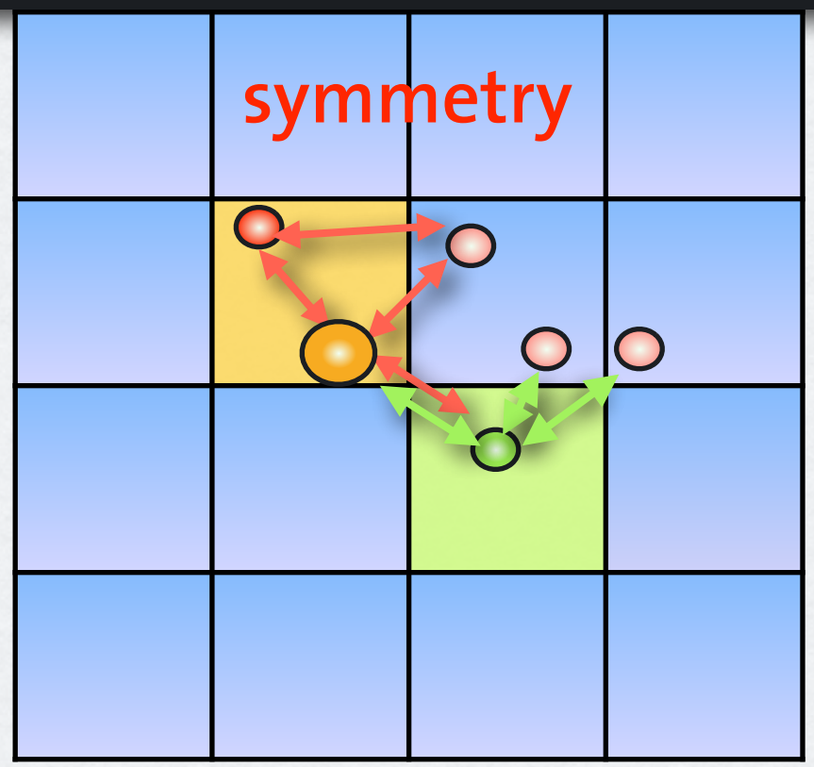
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

- race condition!
- load imbalance (e.g. if some cell is empty)

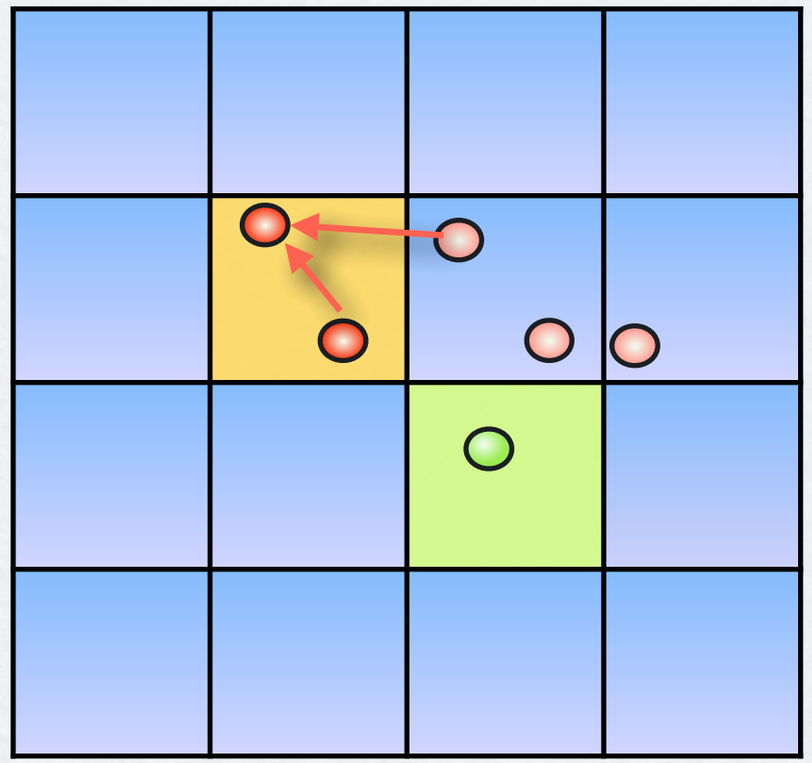


How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task

How to fix race condition:

- ☑ each particle updates just itself



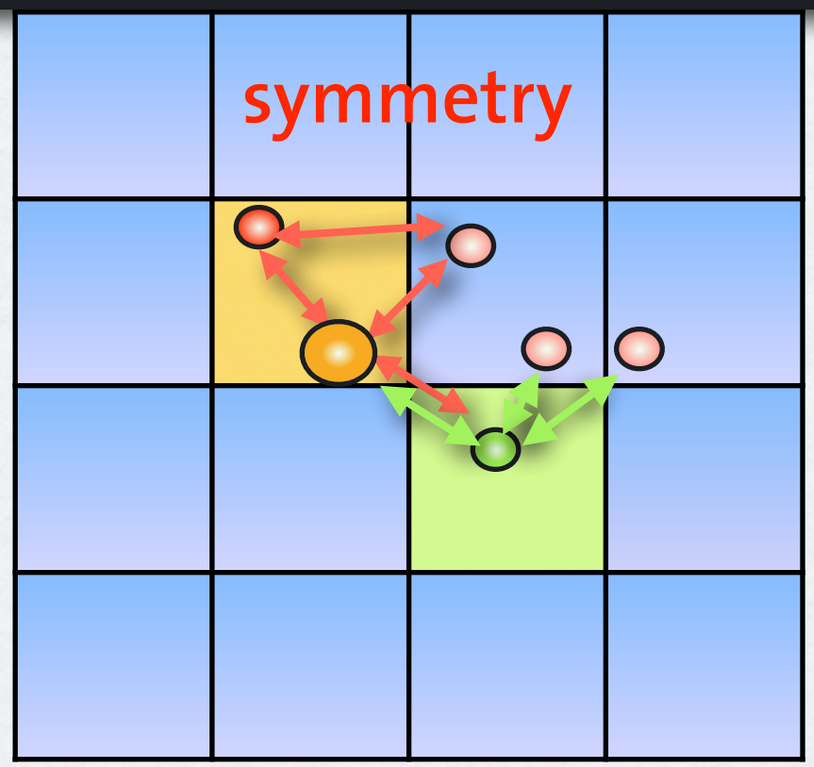
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

- race condition!
- load imbalance (e.g. if some cell is empty)

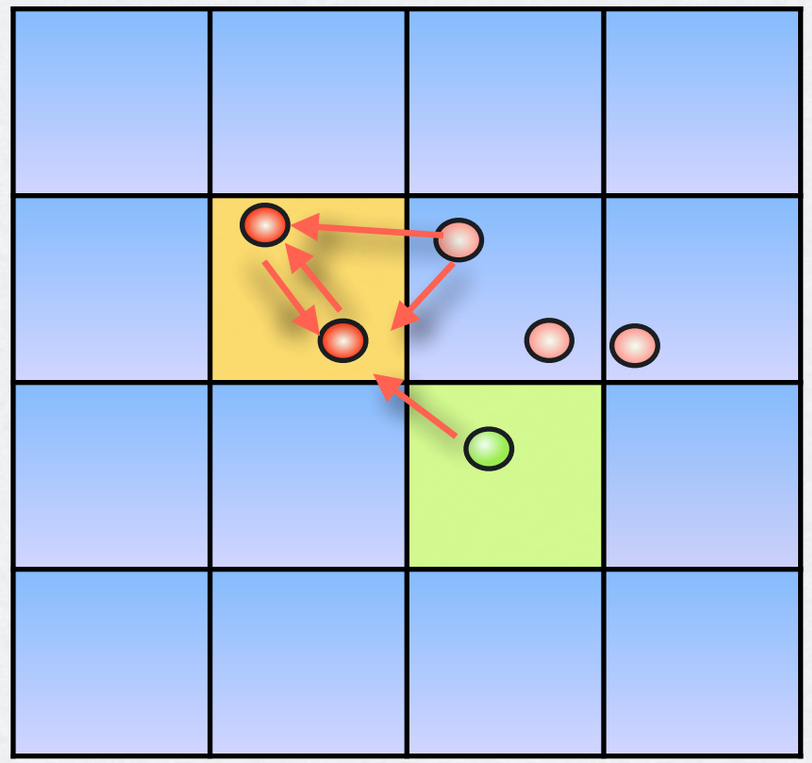


How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task

How to fix race condition:

- ☑ each particle updates just itself



Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

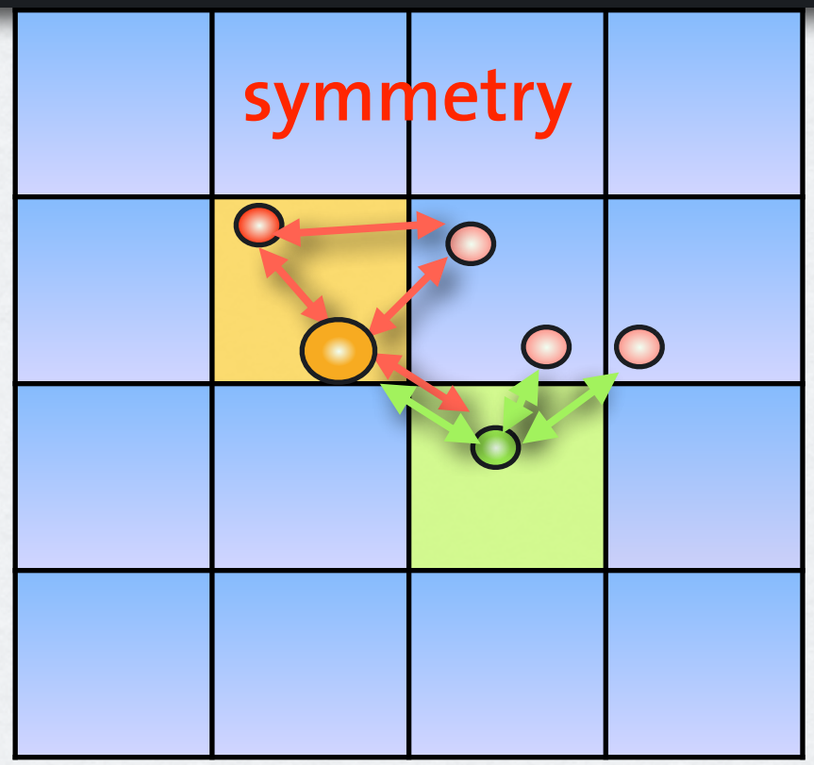
- race condition!
- load imbalance (e.g. if some cell is empty)

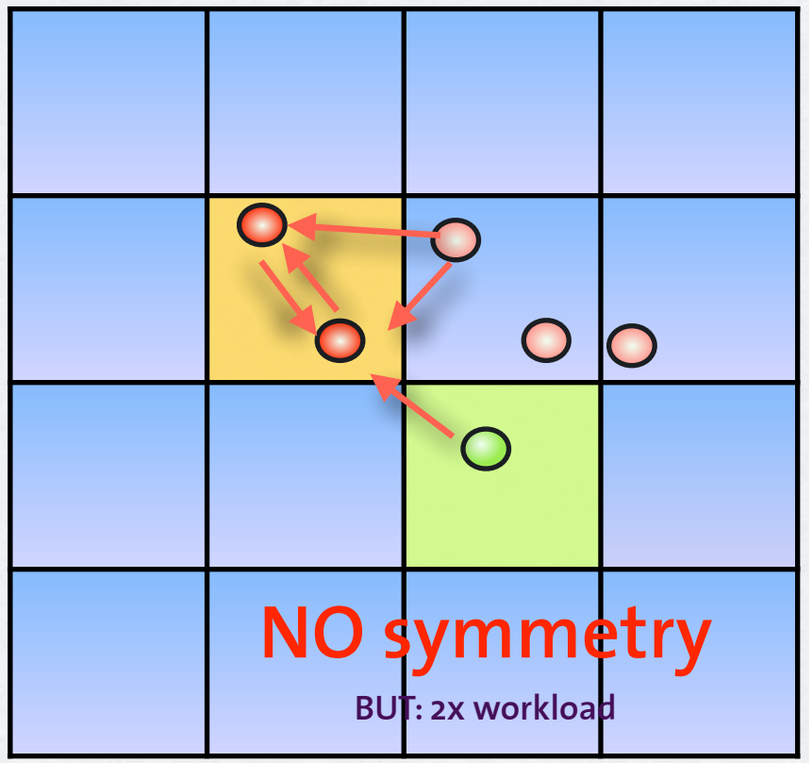
How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task

How to fix race condition:

- ☑ each particle updates just itself





Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

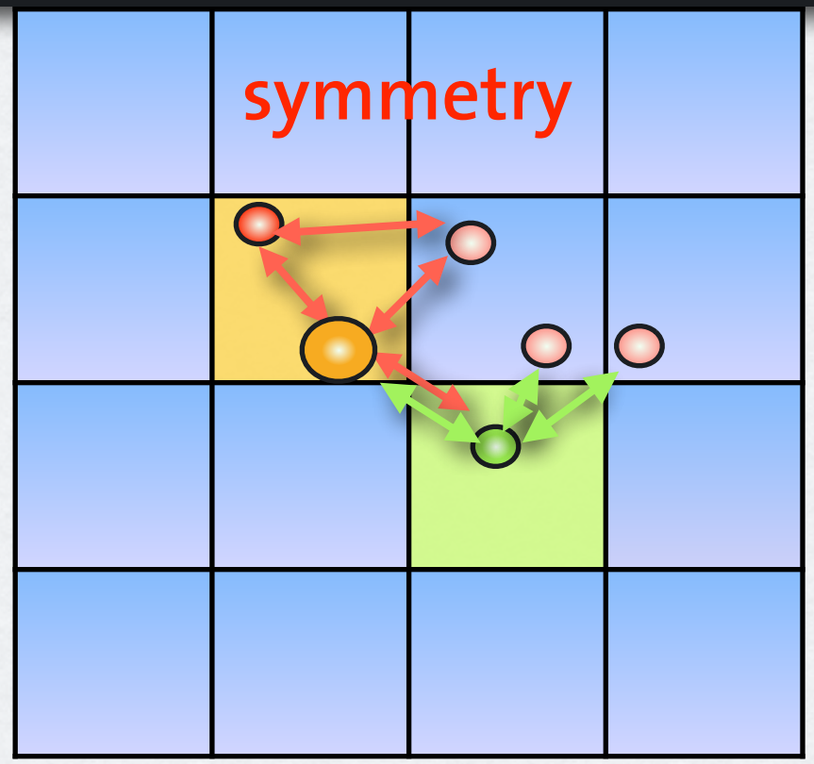
- race condition!
- load imbalance (e.g. if some cell is empty)

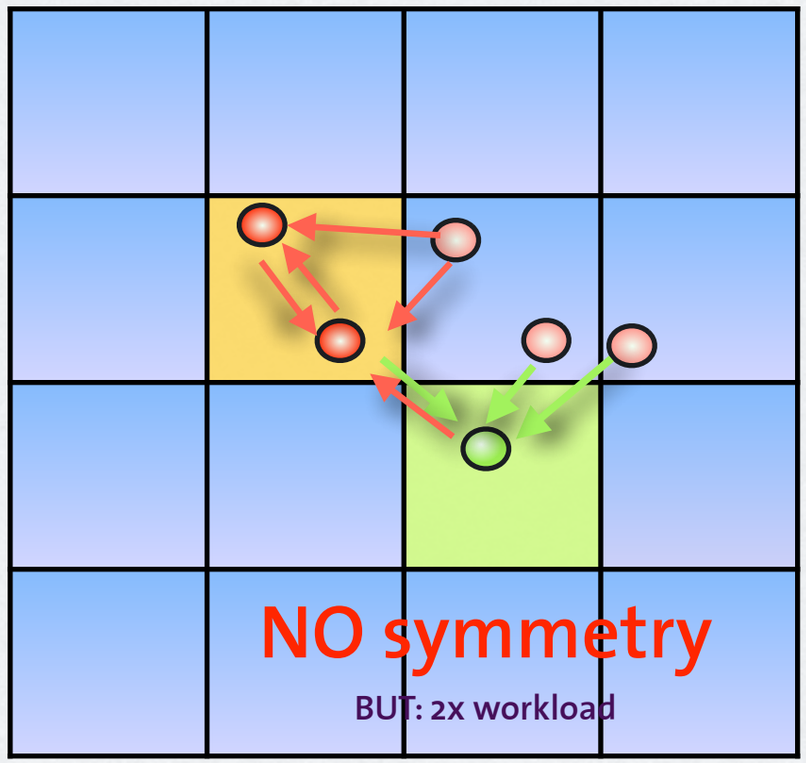
How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task

How to fix race condition:

- ☑ each particle updates just itself





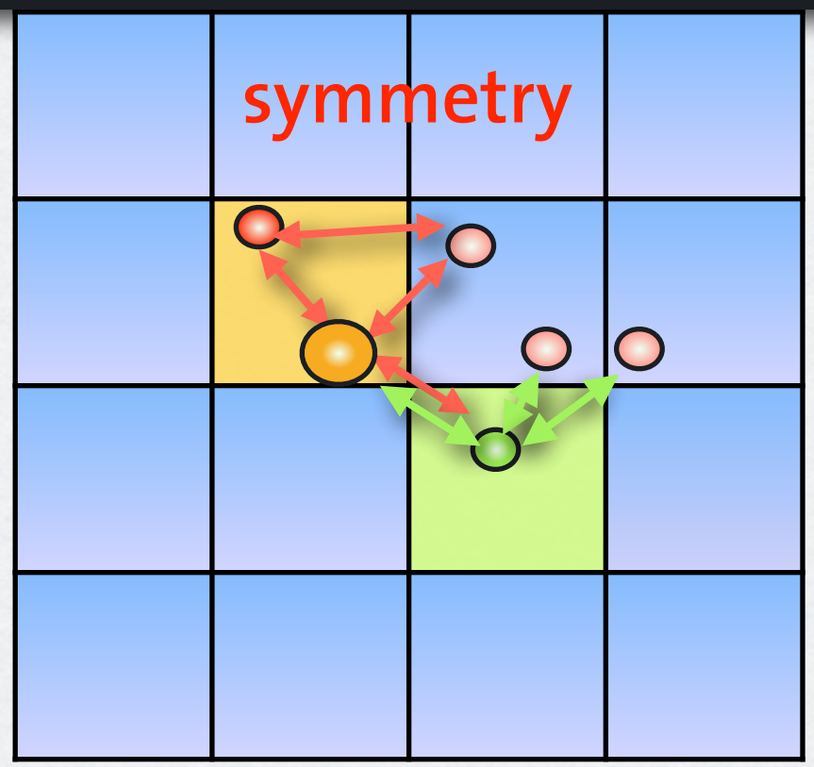
Parallelization over Cells

➡ one thread processes one cell

- particles in cell of thread 1
- particles in cell of thread 2
- particles in other cells

POTENTIAL PROBLEMS

- race condition!
- load imbalance (e.g. if some cell is empty)

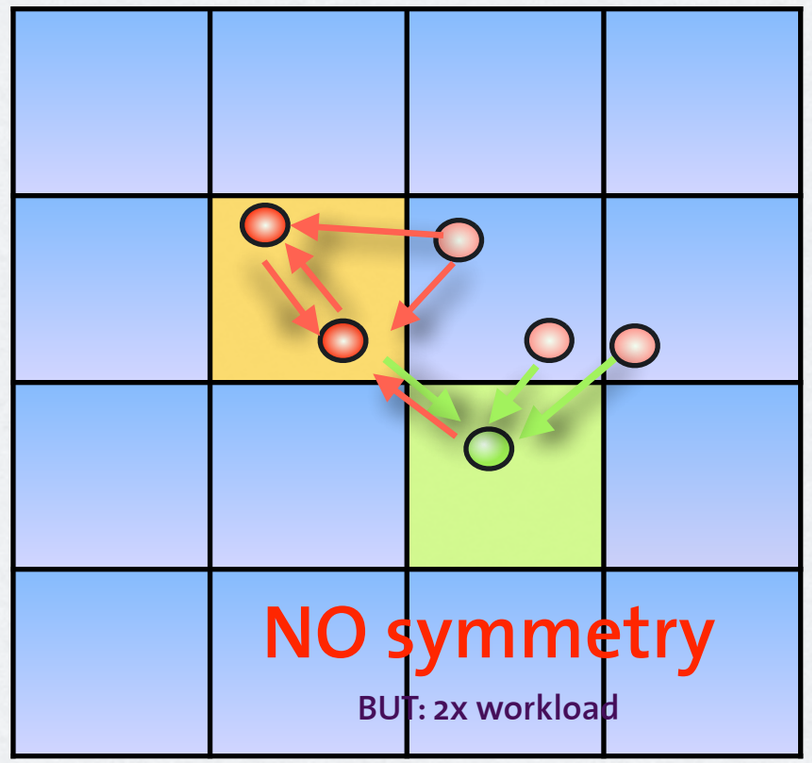


How to reduce load imbalance:

- ☑ skip empty cells
- ☑ TASKING : handling each cell is a task

How to fix race condition:

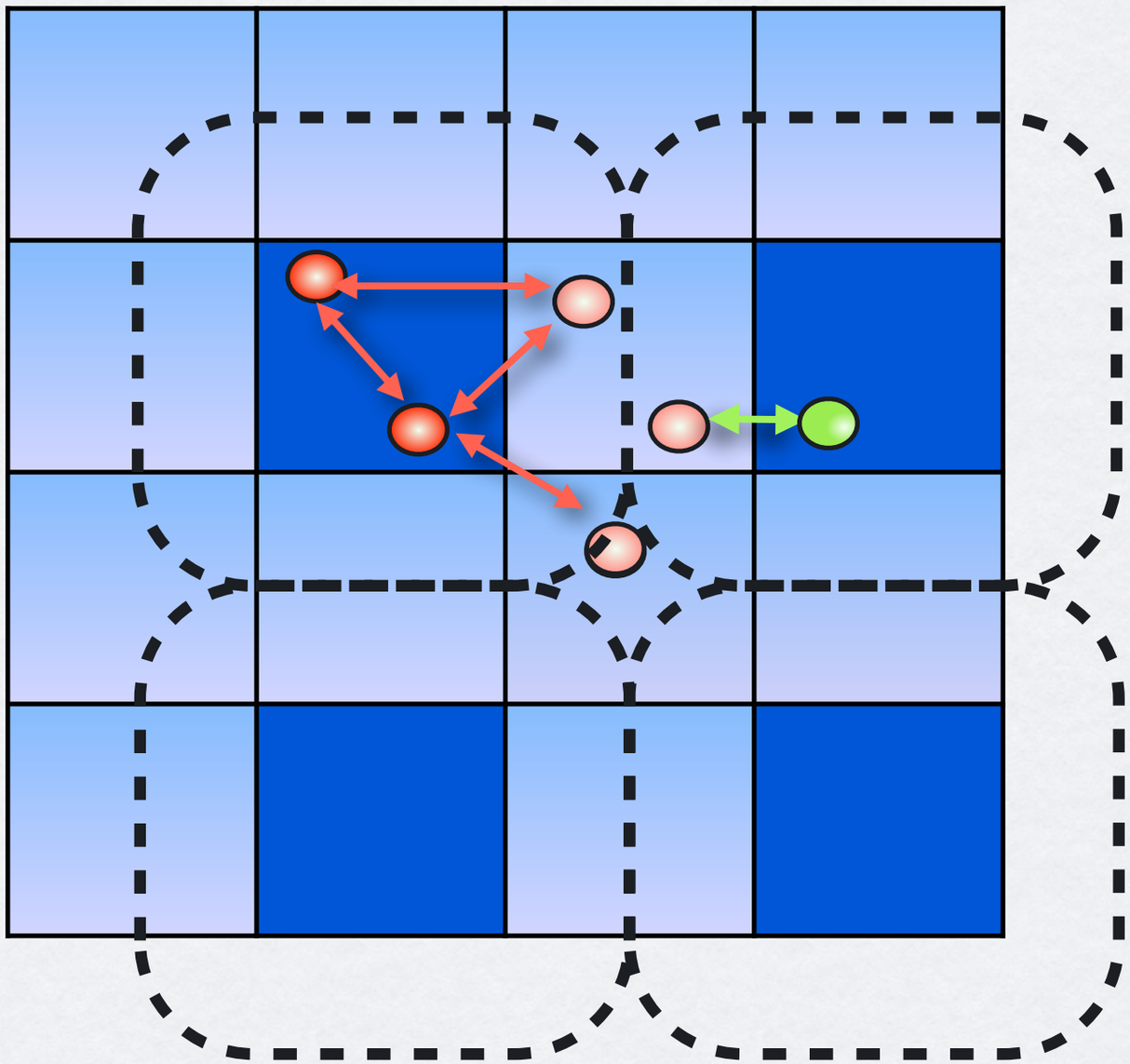
- ☑ each particle updates just itself
- ☑ multipass (next slide)



Divide and Conquer - Multipass

Define Cell size ≧ interaction *diameter* of particles

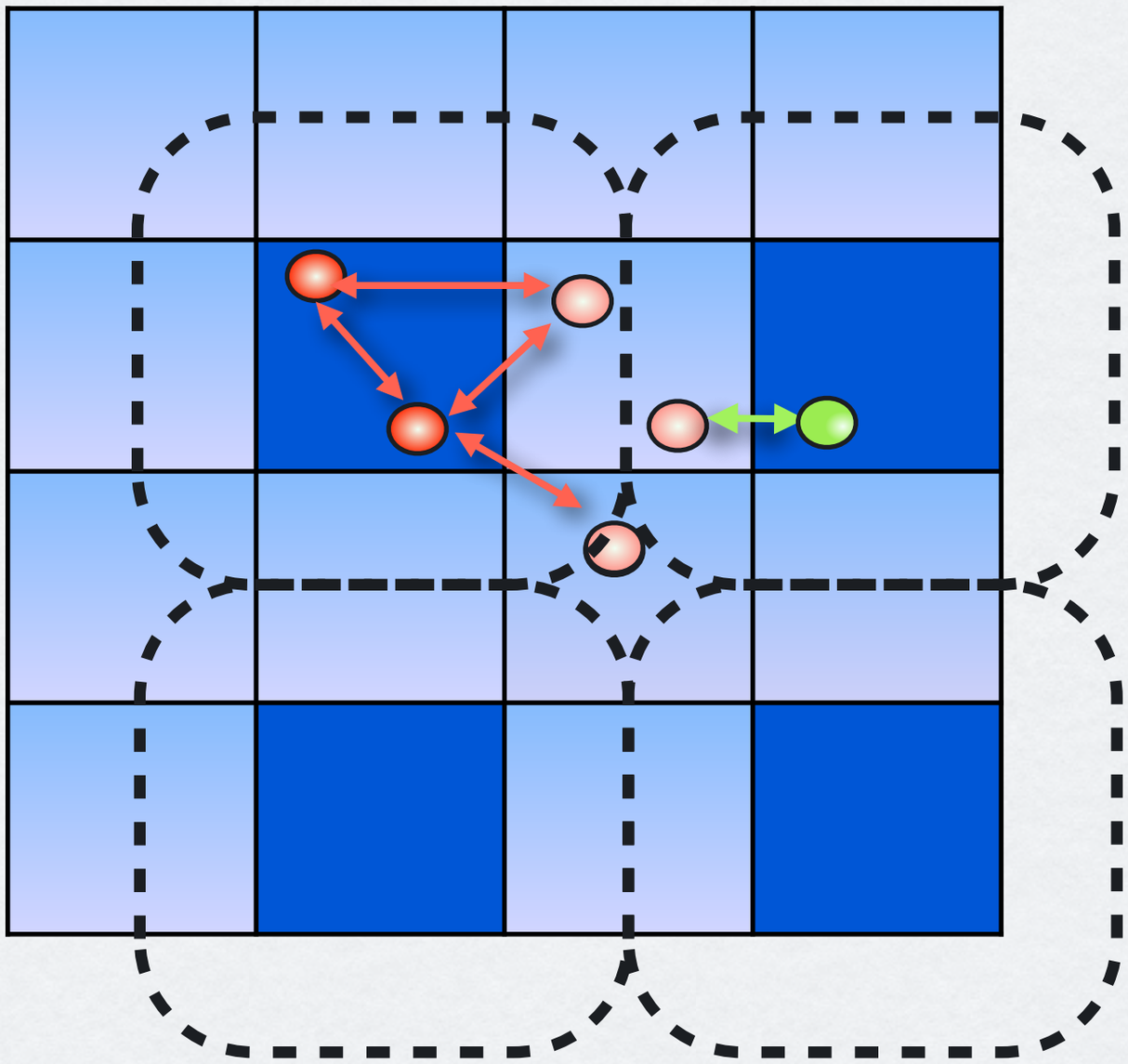
“Pass” : Process non-neighboring cells simultaneously



Divide and Conquer - Multipass

Define Cell size ≧ interaction *diameter* of particles

“Pass” : Process non-neighboring cells simultaneously



☑ **NO race condition**

Divide and Conquer - Multipass

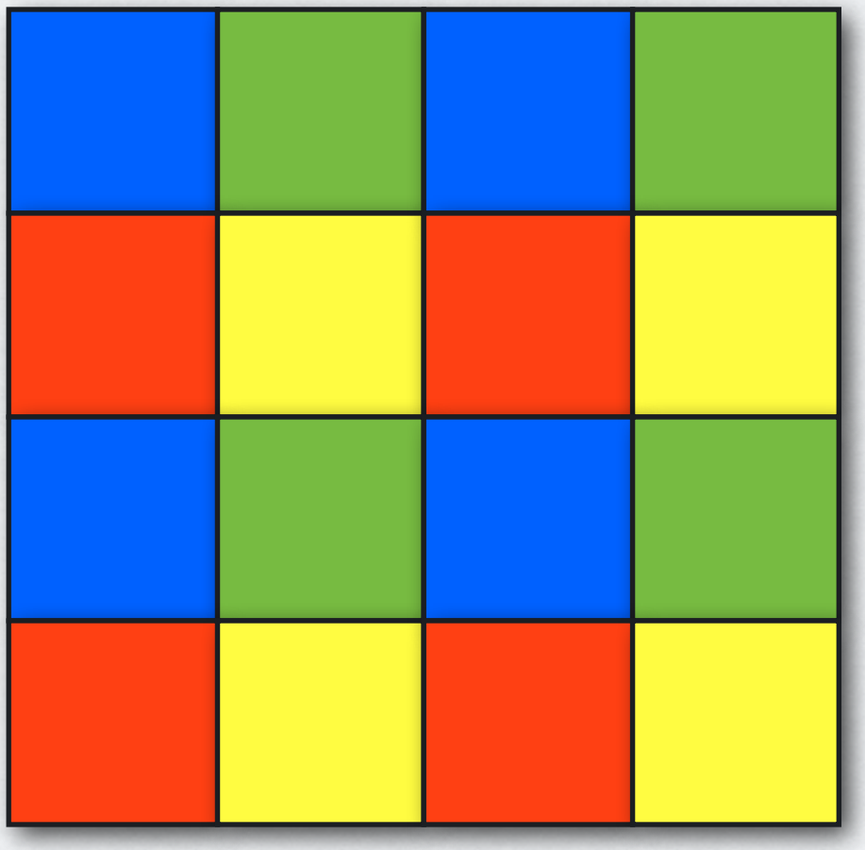
➡ in 2D there must be 4 passes to cover all possible cells

cells of pass 1

cells of pass 2

cells of pass 3

cells of pass 4



cells of the same color can be processed at the same time

➡ in 3D: 8 passes are needed

Issues with Cell Lists

- Curse of dimensionality:
  - 2D - 35% of the particles checked are within the cutoff
  - 3D - 16% of the particles checked are within the cutoff
- Building cell lists every timestep might be expensive