Improving particle communication in IPPL

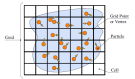
Semester Project for Veronica Montanaro

Paul Scherrer Institute, 10/01/23

Table of contents

Table of contents

The PIC algorithm



Saez et al. (2011)

- Used for problems where particles interact with eachother through fields.
- Main application: plasma modelling (electrostatic or electromagnetic fields).
- Track evolution in phase space, but compute fields on the grid.

The (electrostatic) PIC loop

Table of contents

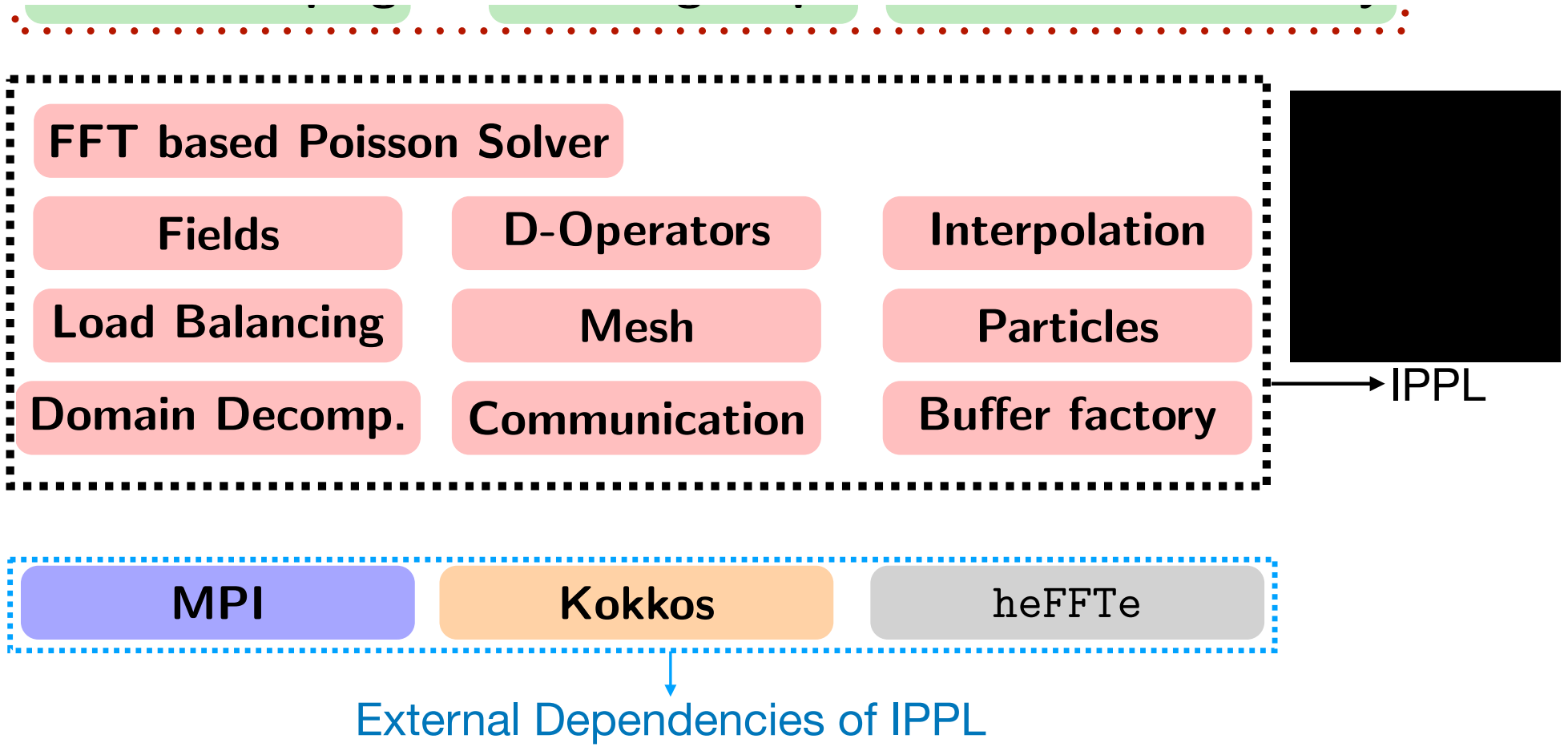
The IPPL framework

- Independent Parallel Particle Layer
- Designed around performance portable and dimension independent particles and fields.
- Part of the OPAL project

The IPPL framework

- Provides a set of mini-apps (Alpine) that makes use of exascale computing to numerically solve some classical problems in plasma physics.
- External dependencies:
  - Kokkos: architecture portability
  - Message Passing Interface (MPI): parallelism
  - HeFFTe: solvers

The IPPL framework



Motivations

- IPPL's main bottleneck is on communication time.
- In some cases, up to 80% of total time.
- Particle communication → main communication cost
- Reducing particle communication cost in IPPL has advantages with and without load balancing.
- **My project**: improving previous communication strategy

Motivations

- Intuition: In electrostatic PIC particles do not travel much more than a cell during a single time step.

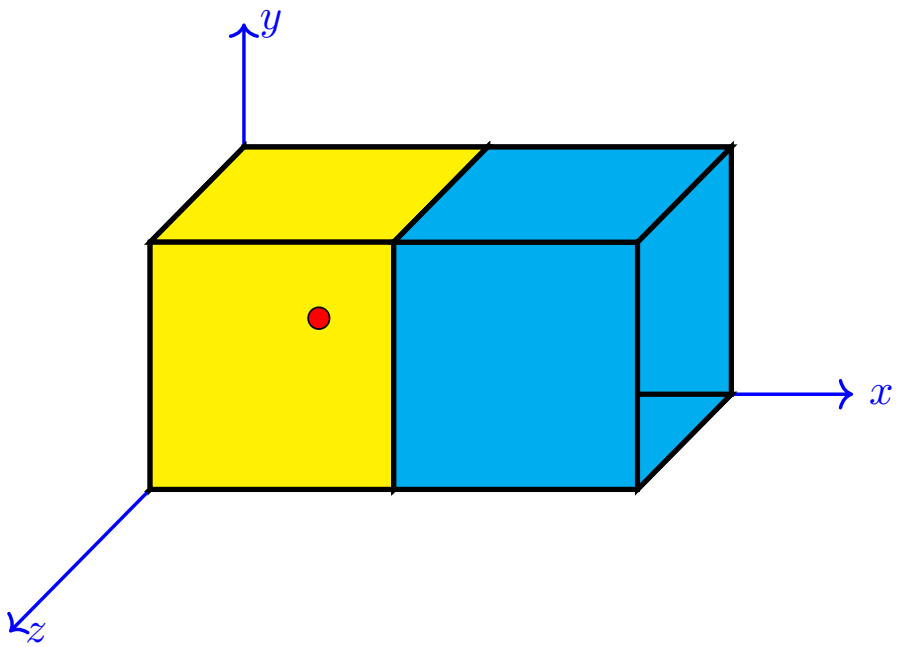


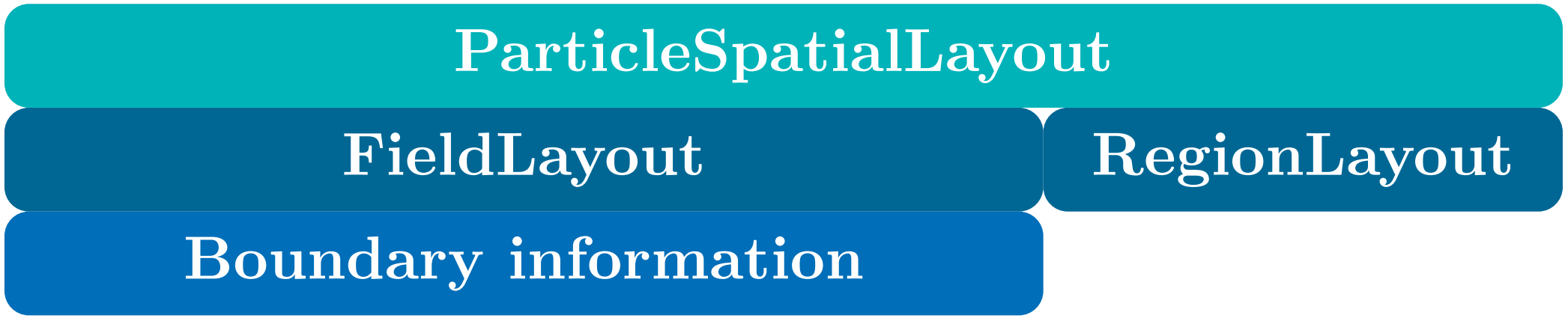
Table of contents

Particle Communication

- Particles are handled through the `ParticleBase` class.
- Particle communication and parallel distribution among processors is done through methods inside the **`ParticleLayout`** class.

ParticleSpatialLayout

- In particular, IPPL 2.0 provides the specialised version **`ParticleSpatialLayout`**.
- Places particles on processors based on their spatial location relative to a fixed grid, based on a Field/Region layout.
- In case of no associated Fiels, a grid is selected based on an even distribution of particles among processors.



ParticleSpatialLayout

- After each timestep, the user must call the `update` routine inside of `ParticleSpatialLayout` to move particles between processors.
- `update` receives as input arguments a view containing the particle information (position and local ID), and the receiving buffer.
- After the Nth call to update, a load balancing routine will be called instead.
- The user may set the frequency of load balancing (N), or may supply a function to determine if load balancing should be done or not.

The ‘update’ routine

Divided in four subroutines:

1. Call `locateParticles` to figure out which particles need to go where.
2. Fill out the send buffers.
3. Delete all the particles that are no longer in the current MPI rank
4. Unpack new particles from receiving buffer.

```
1
2  //update
3
4      //Step 1.
5          /*boolean view for particles ID that
6          moved to other ranks.*/
7          invalid->initialize(nParticles);
8
9          //routine to figure out where particles moved
10         locateParticles( particles, invalid )
11     //Step 2.
12         for each rank{
13             nSend <- count_particles_to_send(rank);
14
15             MPISendBuffer->initialize(nSend);
16
17             MPI_send()MPISendBuffer);
18             }
19
20     //Step 3.
21         for each particle{
22             if( invalid[particle->particleID] == true )
23                 delete_particle();
24             }
25
26     //Step 4.
27         MPI_recv( receiveBuffer );
28
29
```

locateParticles

```
8
9   //routine to figure out where particles moved
10  locateParticles( particles, invalid )
11
```

One of the main bottlenecks

The reason for it is that particle-processor mapping is implemented with a search for all the local particles across all the MPI ranks.

Does not scale!

locateParticles

```
8
9   Kokkos::parallel_for(
10      "ParticleSpatialLayout::locateParticles()",
11      mdrange_type({0, 0},
12                   {ranks.extent(0), Regions.extent(0)}),
13      KOKKOS_LAMBDA(const size_t i, const view_size_t j) {
14
15          bool xyz_bool = false;
16
17          xyz_bool = ((positions(i)[0] >= Regions(j)[0].min()) &&
18                      (positions(i)[0] <= Regions(j)[0].max()) &&
19                      (positions(i)[1] >= Regions(j)[1].min()) &&
20                      (positions(i)[1] <= Regions(j)[1].max()) &&
21                      (positions(i)[2] >= Regions(j)[2].min()) &&
22                      (positions(i)[2] <= Regions(j)[2].max()));
23
24
25          if(xyz_bool){
26              ranks(i) = j;
27              invalid(i) = (myRank != ranks(i));
28          }
29  });
30
31
```

Table of contents

Nearest Neighbor search

- Idea: limit the search to the particle's nearest neighbors exploiting the electrostatic assumption.

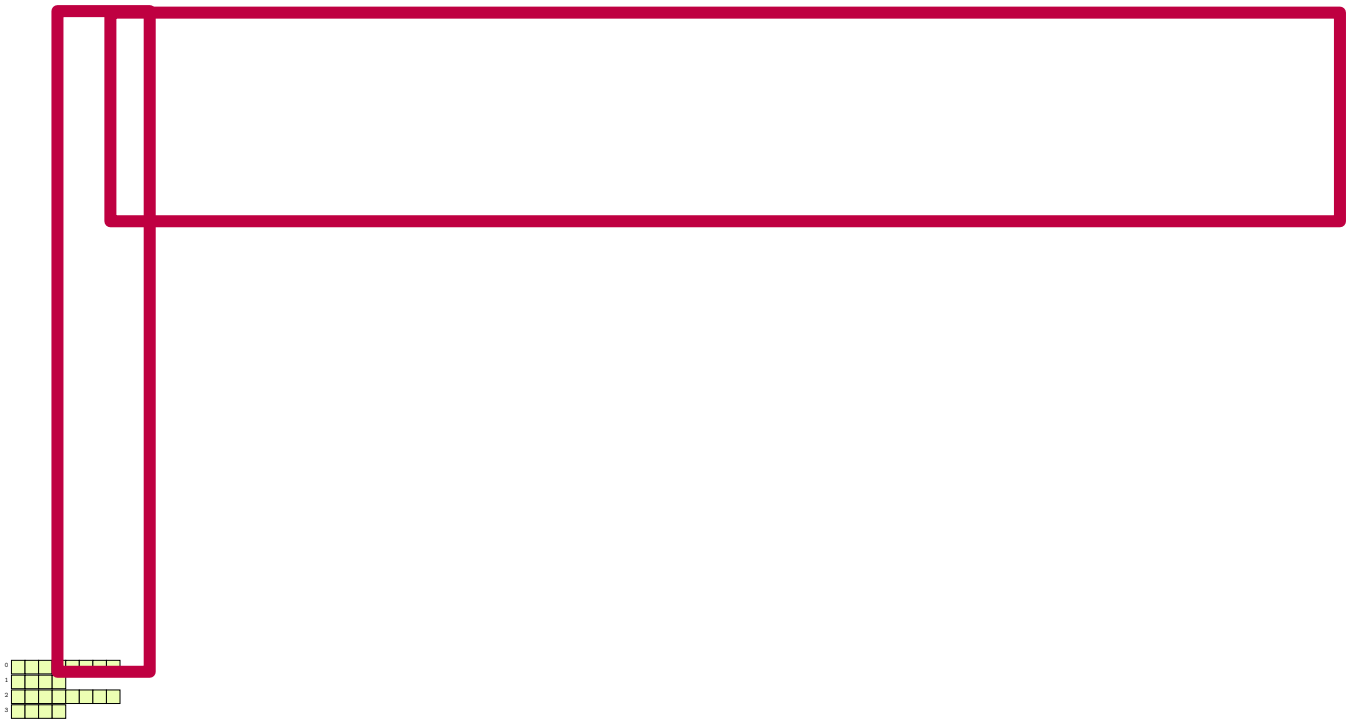
Old	**New**
runtime grows with domain dimension	runtime is constant, with constant $\propto Dim$

- Make use of the neighbor information stored inside `FieldLayout`
- Three containers for **face**, **edge** and **vertex** neighbors, in a Moore-like neighbor scheme.

Structure of neighbors containers

Example: `faceNeighbors` in 2-D, current rank has 8 neighboring ranks in the x-direction and 4 in the y-direction

- First dimension: constant, corresponds to number of neighbors in space
- Second dimension: depends on domain decomposition, contains IDs of MPI ranks sharing the i-th neighbor.



Structure of neighbors containers

- Neighbors containers are currently implemented as C++ `std::array<std::vector<int>>` s.

Cannot be plugged directly into a `Kokkos::parallel_for` loop, as `std::array<>` can't be allocated on GPU

- They need to be converted into a suitable data structure.

Structure of neighbors containers

My solution is possible because of IPPL's only choice of domain decomposition, it being Orthogonal Recursive Bisection $\rightarrow$ Only one neighboring MPI rank per direction.

Structure of neighbors containers

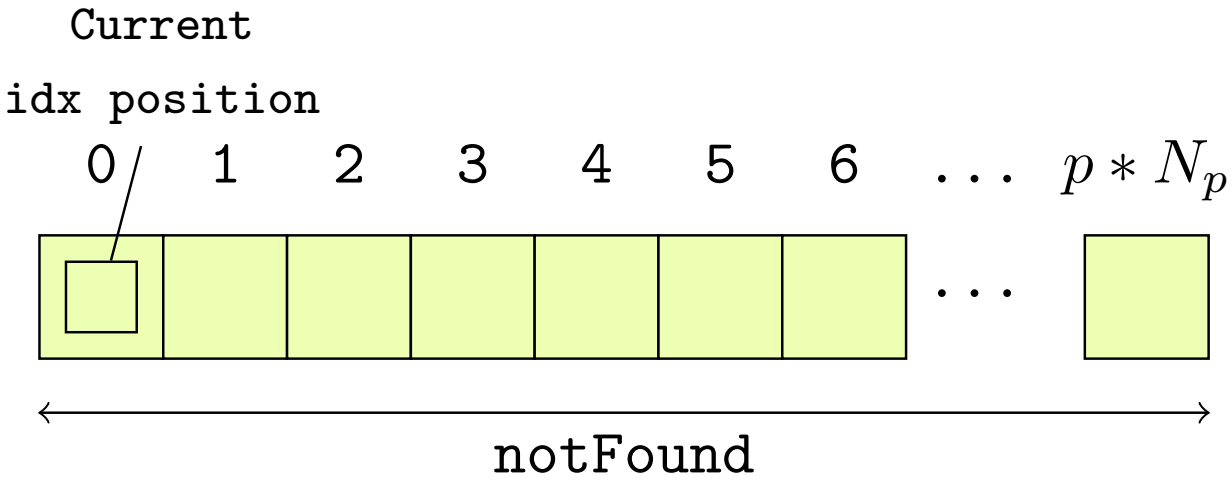
- Making everything into a View doesn't work → initialization has to be done in parallel.
- My choice: flattening the arrays into a **`Ippl::Vector`** → Vector class used for vector fields and particle attributes like the coordinate.
- `Ippl::Vector` is a templated class which wraps data in C vectors, which fixes the allocation problem.
- With the current Kokkos release, jagged columns can't be handled, so right now no solution for more complex neighbor structures.

Handling ‘stray’ particles

- For very fine grids, or for some non-physical tests, the electrostatic assumption may be violated.
- Added an helper Kokkos view (`notFound`) containing the IDs for particles that travel more than one grid cell.
- Dimension of the View is $p*N_{particles}$, $p \in (0,1]$.
- Good choice for $p : 0.3 \rightarrow$ works both for physical and non-physical tests without giving memory overhead

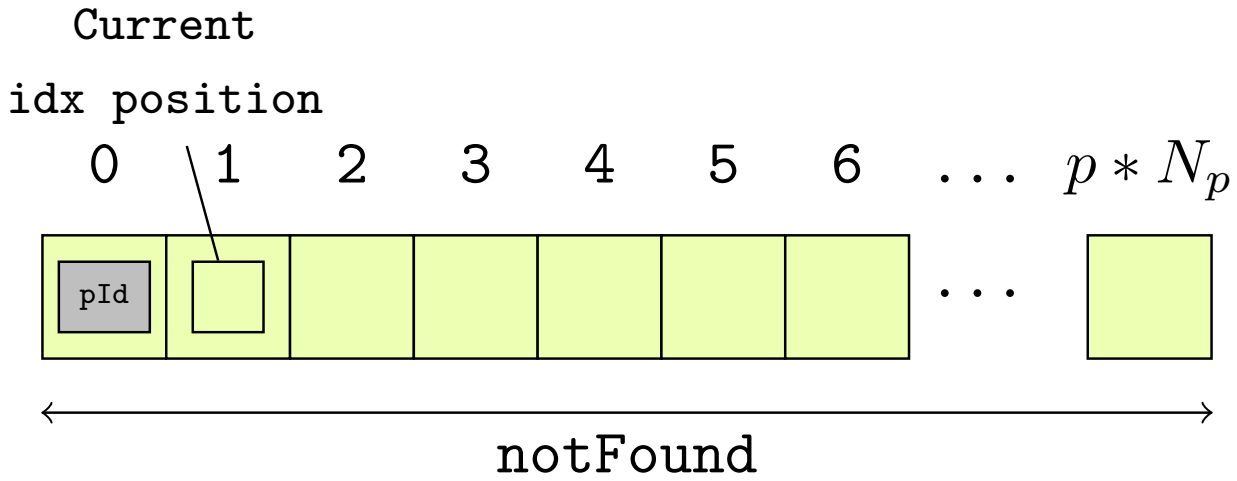
Handling ‘stray’ particles

- An integer index increments everytime that a particle is not in the neighbor set
- For every increment plug the ID of said particle into the helper view `notFound`.
- Race condition avoided by `Kokkos::parallel_scan`’s boolean variable `final`



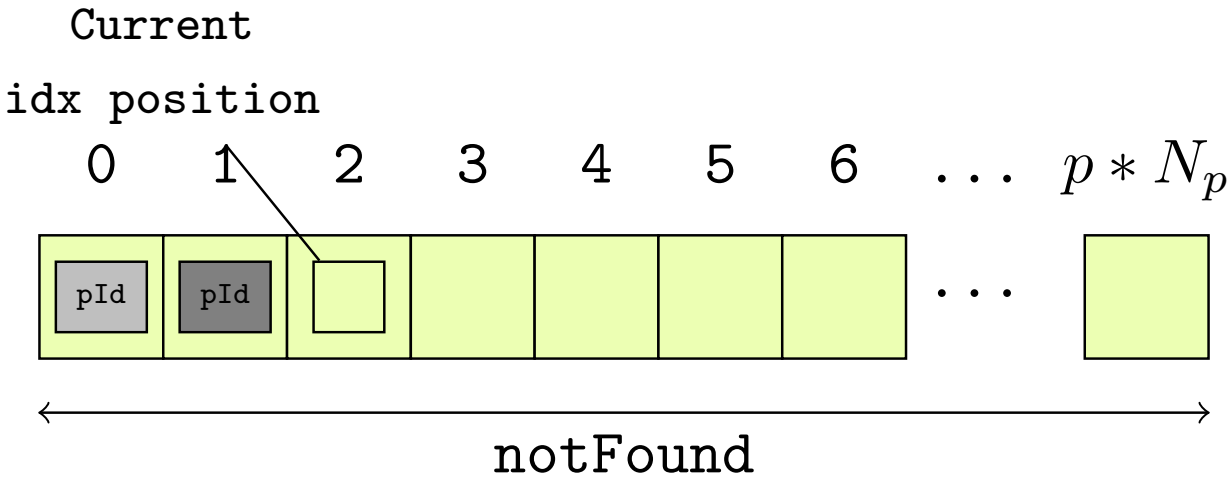
Handling ‘stray’ particles

- An integer index increments everytime that a particle is not in the neighbor set
- For every increment plug the ID of said particle into the helper view `notFound`.
- Race condition avoided by `Kokkos::parallel_scan`’s boolean variable `final`



Handling ‘stray’ particles

- An integer index increments everytime that a particle is not in the neighbor set
- For every increment plug the ID of said particle into the helper view `notFound`.
- Race condition avoided by `Kokkos::parallel_scan`’s boolean variable `final`



New implementation: code overview

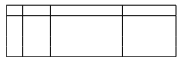
New implementation: code overview

Table of contents

Setup

- Benchmarking: strong scaling analysis on LandauDamping in weak regime, one of the Alpine miniapps.
- Excellent candidate for verification and performance study, because of the availability of analytical results.

Test cases:



Results for case D

GPU benchmarking results

Case D

Case C

Table of contents

Conclusions

- While on CPU the improvement is almost negligible, the new implementation shows its full potential on GPU.
- LocateParticles scales as its time doesn't depend anymore on domain size.
- Maximum improvement on overall communication time is reached for the cases with $8*10^9$ particles, where the time/timestep is about 2.6~2.8 seconds faster than the case without limited search.
- None of the tests fell into the case of stray particles, but that step is still needed for particles with initial uniform distribution.

Future work

- Passing the size of the `notFound` view as a user parameter.
- Finding a way to deal with the neighbor containers in the case of more complex neighbor structures, maybe exploiting the new features of Kokkos 4.0, that will provide tools for 2-D `Kokkos::Vector` and `Kokkos::View`.
- The huge difference in `locateParticles` is translated only partially to `updateParticles`. The other bottleneck may be the `SendPreProcess` loops set, and something alike may be done for that part.

Should i put a last slide for Q&A and thanks for your attention?
Have a beautifully rendered 3D capy