Improving particle communication in the Independent Parallel Particle Layer

Veronica Montanaro
Advisor: Andreas Adelmann
Scientific Collaborators: Sonali Mayani, Sriramkrishnan Muralikrishnan, Matthias Frey

February 7, 2023

Abstract

Simulations of particle dynamics in accelerators benefit largely from High Performance Computing (HPC) resources and from exascale computing. For this reason, we need to adapt the simulations to be ran in parallel and optimize them to be efficient. This is true especially in terms of inter-processor communication, because if not well optimized it's the part that most likely may affect the computational cost. We focus on the Independent Parallel Particle Layer (IPPL), where one of the main bottlenecks is currently on communication of particle information between MPI ranks. In this report we present a very simple yet tricky to implement improvement to the particle communica tion strategy in IPPL. Exploiting a "theoretical" characteristic of explicit PIC, and adapting the previous code to ensure hardware portability, we will show how the new code ends up improving the function by orders of magnitude.

1 Introduction

The availability of tools to simulate the behaviour of charged particles in plasmas and accelerators is extremely important to predict experimental results. Algorithms that track the phase-space evolution of the particles in time have been particularly crucial for the purpose, the most used case being Particle In Cell (**PIC**). Since the size for high fidelity simulations is usually orders of magnitude big, those codes heavily rely on HPC resources, running in parallel on a high number of cores. A lot of research has been carried out in the last years to adapt simulation codes to make it suitable for parallel computing and exascale. In particular, most of the focus is on ensuring machine portability and reducing computational cost.

1.1 OPAL and the IPPL framework

The library we will take into analysis is the Independent Parallel Particle Layer 2.0 (**IPPL 2.0**), the new version of IPPL 1.0. The latter represents the backend of the Object Oriented Particle Accelerator Library (**OPAL**). OPAL is a parallel open source tool, written in C++, for charged-particle optics in linear accelerators and rings, including 3D space charge[5, 4]. IPPL contains the implementation of the main data structures (particles, meshes) and operations on them to do PIC codes[2]. Parallel communication within IPPL is handled through the Message Passing Interface (**MPI**). IPPL 2.0 is currently still in development thus it's not integrated into OPAL yet. The library's main new design features with respect to version 1.0 are performance portability and dimension independence of particles and fields[9].
Portability is ensured by **Kokkos**[7], a library that automatically maps algorithms and data structures to different target architectures, making the code completely hardware-agnostic. Kokkos can also execute parallel operations (such as `parallel_for`, `parallel_scan` and `parallel_reduce`) which work in a similar way as OpenMP parallel regions[3]. Most of the classes in IPPL uses Kokkos data structures called Views. Views are multi dimensional arrays which layout is chosen at compile time to match the computer architecture[3], and therefore they are suitable for access inside of Kokkos' parallel regions, independently on the device's hardware. IPPL also provides a set of mini-apps (Alpine) that makes use of exascale computing capabilities to numerically solve some classical problems in plasma physics (such as Landau Damping or Electron dynamics in a charge neutral Penning trap) with a PIC scheme in a parallel enviroment[9].

1.2 Motivation

Like most PIC codes, IPPL's main bottleneck is communication time, which dominates as the number of MPI ranks increase. In some cases, communication time in IPPL can overcome the computation time. For example, by analysing the results in [9] it can be observed that for some settings of a Landau Damping simulation on GPU the communication time can represent up to

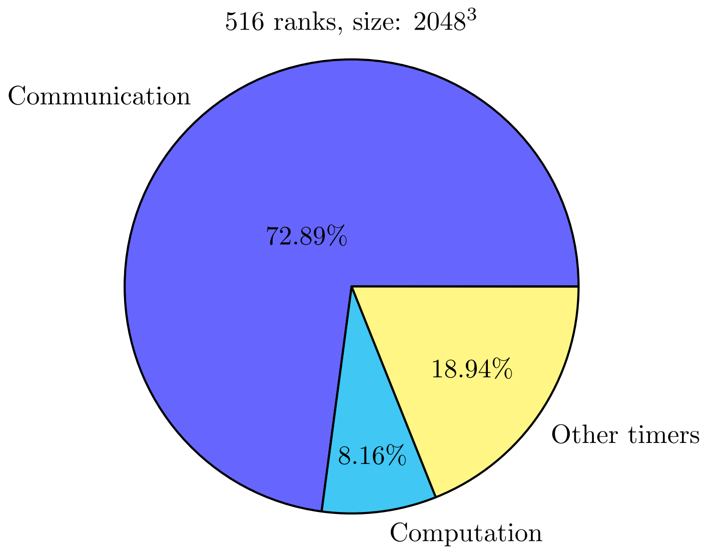


Figure 1: Portion of work for a simulation of the weak Landau Damping problem in the case of 2048 MPI ranks on GPU (for hardware details see [6]) with grid size 2048 cubed. The label *Other timers* refers to kernels like data dumping or warmup.

approximately 70% of the total run time, as shown in Figure 1. In the sequel, we will show an improvement to the communication strategy currently used in IPPL. The new implementation's runtime doesn't grow with the number of MPI ranks, and manages to be independent of it (except for some very rare cases, although later we will explain how this is so rare doesn't cancel the implementation's benefit). The key of the new approach is taking advantage of the fact that in explicit PIC particles do not travel much more than a cell during a single time step, see Figure 2.

2 Implementation

2.1 Particle communication in IPPL

Particles are handled through the `ParticleBase` class (Figure 3), which include classes both for particle attributes and for their communication. Particle communication and parallel distribution among processors is done through methods inside the `ParticleLayout` class. In particular, IPPL 2.0 provides a specialized version of `ParticleLayout`, the class `Particle`

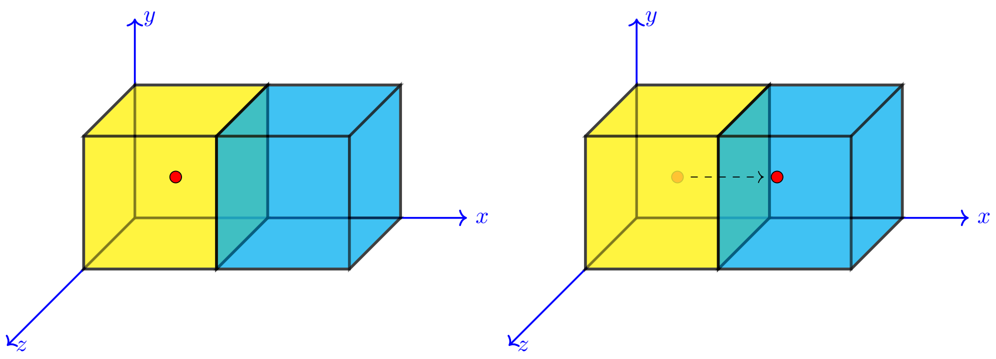


Figure 2: Graphical representation of a PIC particle moving to the neighboring cell after one timestep.

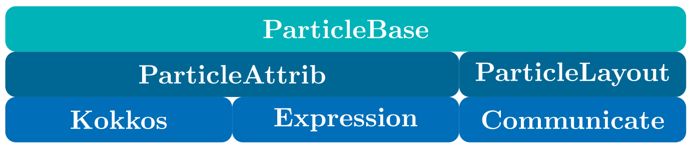


Figure 3: Structure of the ParticleBase class in IPPL.

`SpatialLayout`(Figure 4). Said class places particles on processors based on their spatial location relative to a fixed grid[1]. In particular, this can maintain particles on processors based on a specified `FieldLayout` or `Region Layout`, so that particles are always on the same MPI rank as the rank containing the Field region to which they are local. This may also be used if there is no associated Field at all, in which case a grid is selected based on an even distribution of particles among processors[1]. After each time step in a calculation, which is defined as a period in which the particle positions may change enough to affect the global layout, the user must call the `update` routine, which receives as input arguments a view containing the particle information (position and local ID), and an MPI buffer to receive particles from other MPI ranks. `update` will move particles between processors. 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

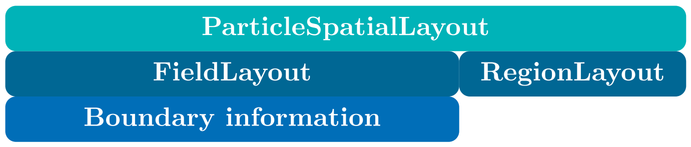


Figure 4: Structure of the ParticleSpatialLayout class in IPPL.

to determine if load balancing should be done or not.

2.1.1 update and locateParticles

The `update` routine is divided in four subroutines: First, it calls the `locateParticles` routine to figure out which particles need to go where. After the mapping is done, it fills out the send buffers with the particles that travelled to other MPI ranks, and sends them. As a final step, it deletes all the particles that are no longer in the current MPI rank (labeled as `invalid`) and unpacks the new particles from the receiving buffers. The procedure is described in detail in Algorithm 1.

```
//update

//Step 1.
    /*boolean view for particles ID that
    moved to other ranks.*/
    invalid->initialize(nParticles);

    //routine to figure out where particles moved
    locateParticles( particles, invalid )

//Step 2.
    for each rank{
    nSend <- count_particles_to_send(rank);

    MPISendBuffer->initialize(nSend);

    MPI_send(MPISendBuffer);
    }

//Step 3.
    for each particle{
    if( invalid[particle->particleID] == true )
    delete_particle();
    }

//Step 4.
    MPI_recv( receiveBuffer );
```

Algorithm 1: `update`'s workflow.

The focus of the research was on `locateParticles` (line `9` of Algorithm 1), as it is one of the main bottlenecks in the routine. In the current implementation the particle-processor mapping is implemented with a search for all the local particles across all the MPI ranks (see Algorithm 2), which does not scale.

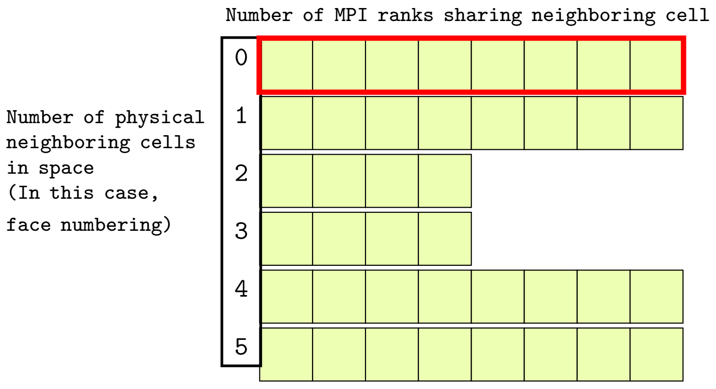
```
//locateParticles

//begin parallel loop
   Kokkos::parallel_for(
            mdrange_type({0, 0},
                {particles.size(), Regions.size()}),
            KOKKOS_LAMBDA(const size_t i, const size_t j){

                found = false;

                found = particle(i)->is_in_region(Regions(j))

                if(found==true){
                    ranks(i) = j; //array that maps particle i to rank j
                    invalid(i) = (j != myRank);
                }

            }
```

Algorithm 2: Parallel loop contained in `locateParticles`.

2.2 Methodology

2.2.1 Nearest neighbor search

The idea was to limit the search to the particle's nearest neighbor exploiting the explicit PIC assumption explained in section **1.2**. Neighbor information is stored inside of the `FieldLayout` class, which includes three containers for it, respectively for information on face, edge and vertex neighbors. The structure for these containers is the one of a 2-D array with jagged columns. The first dimension is fixed and it's proportional to the spatial dimension, while the second is dynamically allocated and depends on the domain decomposition technique. An example can be seen in Figure 5. This is currently implemented



Number of MPI ranks sharing neighboring cell
Number of physical neighboring cells in space (In this case, face numbering)
0
1
2
3
4
5

Figure 5: Visualization of the structure for face neighbors in 3-D, assuming a domain decomposition with 2^3 neighboring MPI ranks in the x and z direction and 2^2 in the y direction. In this case, the faces are numbered in the following order: lower face in x-direction, higher face in x-direction, lower face in y-direction, higher face in y-direction, lower face in z-direction, higher face in z-direction..

in IPPL with a C++ `std::array<>` with `std::vector<>` of integers as entries. The reason for this structural choice is because in C++, the class `std::vector<>` is particularly suitable for data which dimension is unknown at compile time.

However, the simplest solution of plugging those arrays directly into a Kokkos parallel loop, in the same fashion as the full domain search did with the Regions view, doesn't work. Doing that will lead to no problems on CPU, but won't work on GPU as `std::array<>` cannot be allocated on GPU devices. So it's essential to provide a suitable data type where to transfer the neighbor information. Currently, such containers are defined locally inside of `locateParticles` and data from the neighbor arrays is copied inside of them. The decision was made because the neighbor arrays are both initialized and accessed only in serial in other parts of the code. It is object of discussion whether it would be better for the code's purpose to change the data type of the neighbor arrays. IPPL currently provides uniform domain decomposition, where each MPI rank has only one neighbor per dimension, as shown in Figure 6. This made possible the simplest soultion of flattening the arrays into a 1D `Ippl::Vector`, a vector class native to IPPL used for field quantities or particle attributes, like their location. More in detail, this class is a wrapper for C vectors, which fixes the allocation problem as those can be allocated on device memory.

For domain decomposition techniques with more complicated neighbor

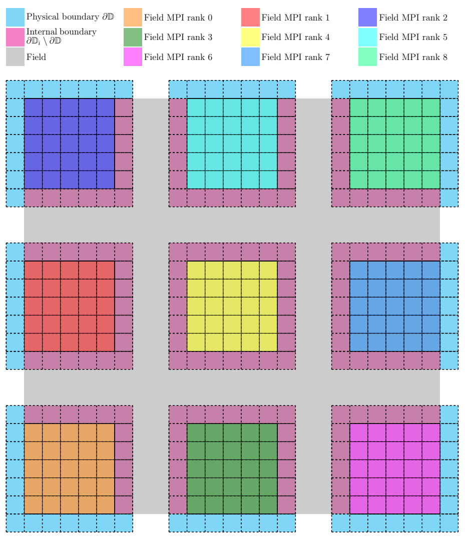


Figure 6: Visualization of uniform domain decomposition.

patterns, the same flattening technique can be used, by implementing a suitable constructor. The reason why we didn't opt for a 2-dimensional `Ippl ::Vector` is the fact that `Ippl::Vector` is allocated statically at compile time. The second dimension of the 3-dimensional (faces) and 2-dimensional (edges) neighbor vectors is dynamically allocated, and therefore having a 2-dimensional `Ippl ::Vector` won't work. Note that transferring all the data inside of a `Kokkos ::View` wouldn't have been a valid solution, as views are always allocated in parallel (because of first-touch policy), and therefore the problem of accessing a C++ `std::array<>` inside of a parallel region presents itself again.

2.2.2 Handling exceptions in moving particles

The code is done under the assumption that PIC particles cover a very small distance per timestep in the computational domain for explicit time-stepping schemes. However, for very fine meshes or for some non-physical tests, there might be exceptions and therefore just one 'out-of-place' particle could lead to the whole simulation to crash. To prevent the case, we create an helper Kokkos view (referred as `notFound`) containing the IDs for particles that travel more than one grid cell, and thus have not been found in the nearest neighbors set. Since the number of particles are usually very high and we want to avoid memory overhead, `notFound` has dimension $p * n_{particles}$, where $p \in (0, 1]$. The best choice for p has been proved to be 0.3, as it works both for physical and non-physical tests without giving memory overhead. Further implementations of IPPL may let p be passed as a simulation parameter. Inside of the parallel Kokkos kernel, an integer variable representing the number of not-found particles is kept and updated every time a particle violating the assumption is found. For every increment, the ID of said particle is then stored inside of `notFound`. If, at the end of the parallel loop, the index is higher than zero, the full domain scan is run again looping over the particles IDs contained in `notFound`. This increment-and-store scheme is possible without any race condition thanks to the Kokkos kernel `parallel_scan`, that works similarly to a `parallel_reduce`. While the latter simply performs a parallel loop with a reduction operation (increment, sum, maximum or minimum, etc.), `parallel_scan` adds an additional boolean value (`final`) that ensures access to a code block in mutual exclusion, and only when all threads have finished their work. To summarise, the new `locateParticles` works in four steps, illustrated in Algorithm 3:

```

Kokkos::parallel_scan(
        Kokkos::RangePolicy<size_t>(0, ranks.size()),
        KOKKOS_LAMBDA(const size_t i, size_t& idx, const bool final) {
            //Step 1 : search in current rank
            found = false;
            found = particle(i)->is_in_region(Regions(myRank))
            if(found==true){
            ranks(i) = myRank;
            invalid(i) = false;
            }

            //Step 2 : search in neighbors set
            else{
                for each face{
                    rank = faceNeighbors_v[face];
                    /*repeat check
                    ....
                    */
                }
                /*repeat for edges and vertices
                ....
                */
            }

//Step 3: check if particle isn't in any neighbor set
            //region to avoid race condition
            if( final && !found ){
                notfound(idx)=i;
                }
            if( !found ) idx+=1;
            }, nLeft); //idx value is loaded into nLeft at the loop's end

    //Step 4: check for stray particles and run loop on whole domain
    if(nLeft > 0) {
    Kokkos::parallel_for(
            mdrange_type({0, 0},
                {nLeft, Regions.extent(0)}),
            KOKKOS_LAMBDA(const size_t i, const size_t j) {
            pId = notfound(i);
            found = false;
            found = particle(pId)->is_in_region(Regions(j))
            if(found = true){
            ranks(pId) = j;
            invalid(pId) = true;
            }
            });
```

Algorithm 3: locateParticle new implementation

3 Benchmarking and Results

3.1 Setup

The benchmarking of the new implementation was done against the previous results available in[9], with the same settings. We performed strong scaling analysis, i.e. fixing the problem size and incrementing the number of cores, on four particle configurations shown in Table 1. The problem selected for the analysis is Landau damping in weak regime, which is one of the Alpine mini-apps. This problem is particularly suitable for benchmarking and checking correctness of the code, because of its well-known properties and the availability of analytical results. The CPU runs were made with $\{4, 8, 16, 32, 64\}$ cores for cases A and C, and with $\{32, 64, 128, 256, 512\}$ cores for cases B and D. On GPU, the tests ran with $\{16, 32, 64, 128.256\}$ cores for cases A and B, and with $\{128, 256, 512, 1024, 2048\}$ cores for cases C and D.

The machine used for the runs is Piz Daint [6].

Case	Grid size (N_c)	Total number of particles(N_p)	Particles per cell (P_c)
A	512^3	1,073,741,824	8
B	1024^3	8,589,934,592	8
C	256^3	1,073,741,824	64
D	512^3	8,589,934,59	64

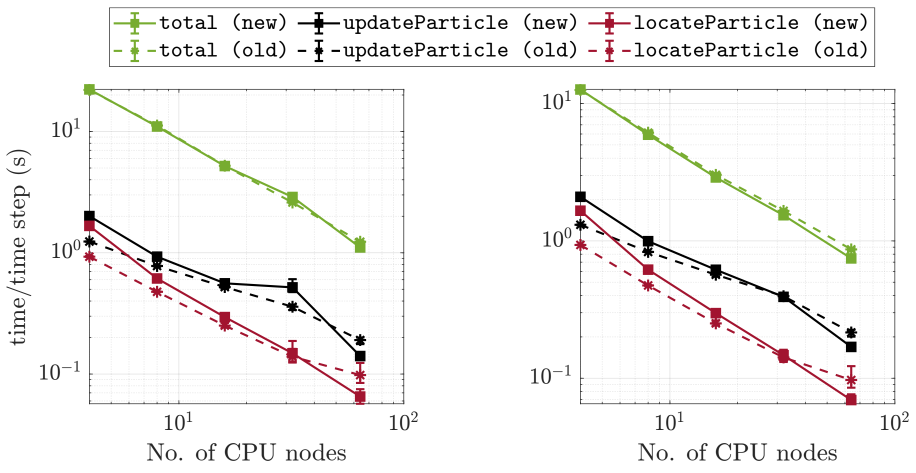
Table 1: Setup for strong scaling.

3.2 CPU strong scaling

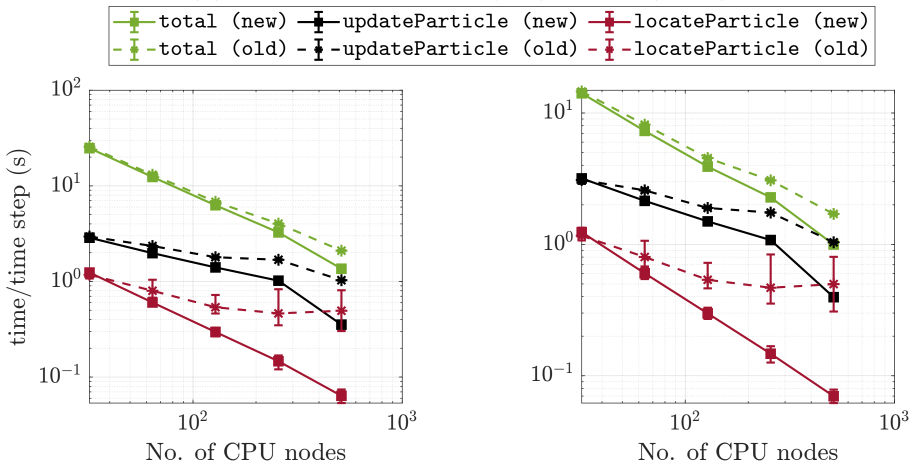
The scaling of `locateParticles` is barely visible in cases A and C (Figure 7a), where the function already scaled. For 4 CPU nodes it even presents some delay with respect to the current implementation, since in that case the total number of neighbors is higher than the total number of MPI ranks. The implementation benefits are seen starting from > 64 nodes. This becomes particularly noticeable in cases B and D (Figure 7b). For a number of cores superior to 100, the current implementation starts suffering from latency, which isn't the case for the new version. In all four tests, the limited search does not seem to affect `update` significantly. As for how the change in `locateParticle` is transferred to the total communication time (Figures 8 and 9), the change is almost negligible.

3.3 GPU strong scaling

Coherently to what has been observed in the CPU analysis, increasing the number of cores make the changes in locateParticle extremely visible, as it can be clearly seen from Figure 10. The best results are given one more time by cases B and D (Figure 10b), but in the GPU runs there is improvement even for the lowest number of cores in cases A and C (Figure 10a). It becomes evident how the new implementation represents an impressive step further, since `locateParticle` ends up being one order of magnitude faster in the run with 256 cores (highest number of cores) for cases A and C, and even two order of magnitudes faster for 2048 cores in cases B and D.
Another good result achieved on GPU is the fact that the overall time spent on communication time is affected way more, as it is lower compared to the current implementation (Figures 11 and 12). For example, in case B (Figure 11b) and D (Figure 12b), the global communication time for 2048 cores has been reduced of about 2.8 times. Moreover, for D, changes in the

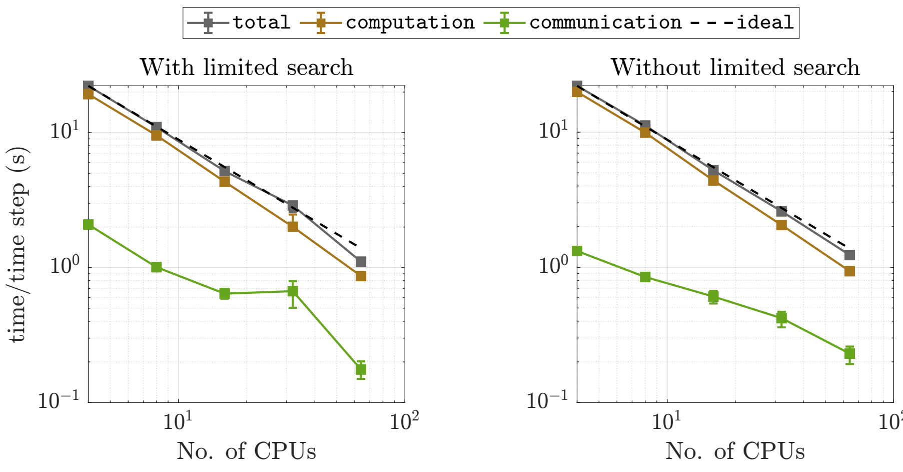


(a) Results for cases A (left) and C (right)

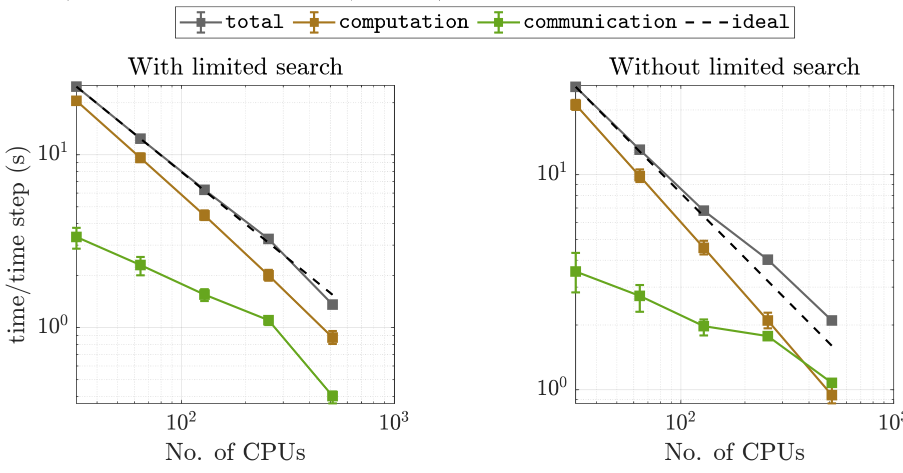


(b) Results for cases B (left) and D (right)

Figure 7: Runtimes on CPU of the functions `locateParticle` and `update` (referred as `updateParticle` in the legend). The curve `total` refers to the total simulation runtime.

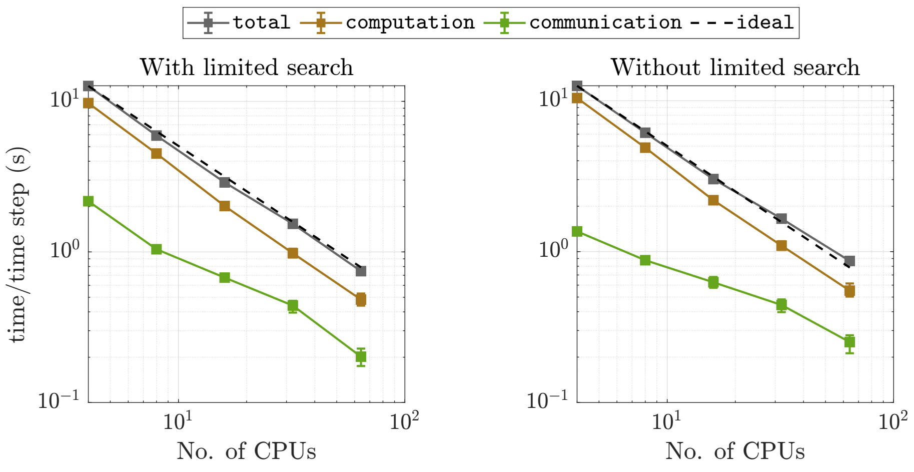


(a) Results for the new implementation (on the left) vs. the current one (on the right), with grid size 512^3 (case A).

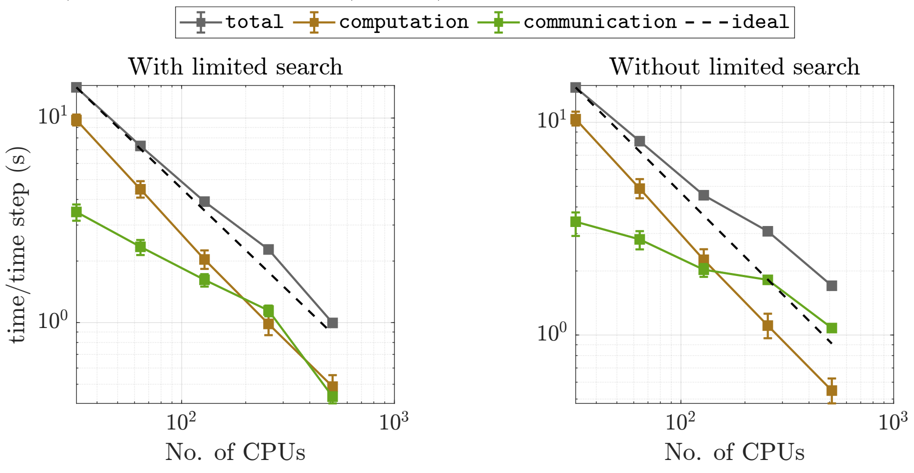


(b) Results for the new implementation (on the left) vs. the current one (on the right), with grid size 1024^3 (case B).

Figure 8: Total time spent on communication and computation for the CPU runs and 8 particles per cell (cases A and B). In the plot, the total runtime is also shown, along with the ideal scaling.



(a) Results for the new implementation (on the left) vs. the current one (on the right), with grid size 256^3 (case C).



(b) Results for the new implementation (on the left) vs. the current one (on the right), with grid size 512^3 (case D).

Figure 9: Total time spent on communication and computation for the CPU runs and 64 particles per cell (cases C and D). In the plot, the total runtime is also shown, along with the ideal scaling.

total runtime become also visible, and we end up having a simulation which in the best case is about 2.1 times faster than the one using the current implementation.

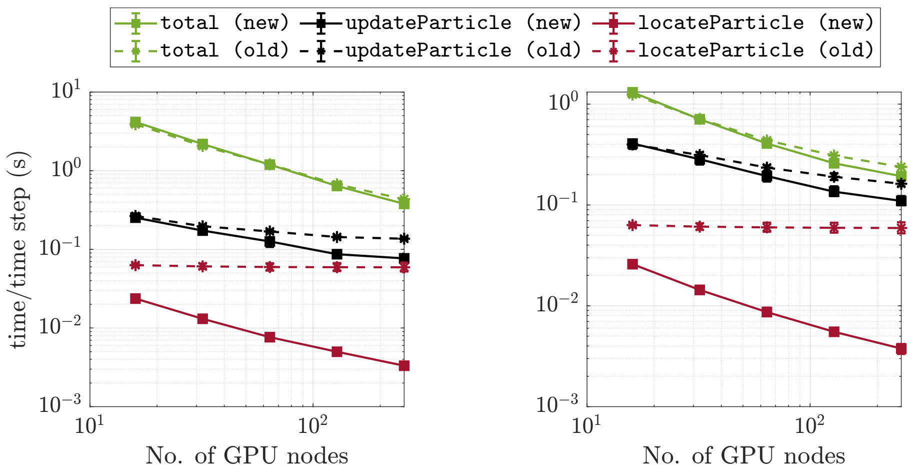
4 Conclusions and future work

4.1 Conclusions

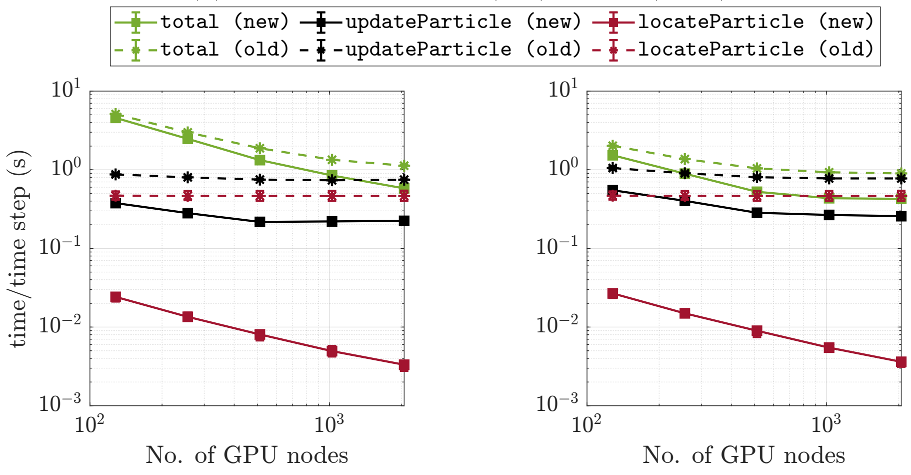
While on CPU only a minimal improvement can be seen, and only after a certain number of cores, the new implementation shows its full potential on GPU. We achieved a runtime scaling of `locateParticles`, which doesn't depend on the number of MPI ranks anymore. The new implementation makes `locateParticles` faster by one to two order of magnitudes, and in the best case scenario the time spent on communication is reduced by a factor of almost 3, and of about 2 when it comes to the total runtime. We chose $p = 0.1$ for the runs, since no exception to the explicit timestepping scheme was found. However, other tests included in the IPPL `test` directory were sensitive to 'stray' particles. In particular, the most affected were `TestScattering.cpp` and `PIC3D.cpp`.

4.2 Future work

Passing the dimension of the `notFound` array as a user parameter is still to be implemented. Moreover, a more efficient way to deal with the neighbor containers in the case of more complex neighbor structures that doesn't involve flattening is still to be found. The Kokkos developer team suggested that using a View with the space `Kokkos:: SharedSpace` can be used for this purpose. This feature will only be available after the release of Kokkos 4.0, which is still in development. With `Shared Space`, it would be possible to create a `Kokkos::View` of Views in page migratable memory, then wrap all the neighbor's inner vector into a host `Kokkos::View` and call a `deep_copy` to copy all data in device memory. This is shown in Algorithm 4. The order-of-magnitude runtime improvement in `locate Particles` is only partially seen in `updateParticles`. The bottle neck now might be related to another subroutine, represented by the step 2 in Algorithm 1 and referenced as `SendPreProcess` in IPPL's timings. This subroutine contains a similar set of loops as the one causing the bottleneck in `locate Particle`, as it scans all the MPI ranks to pack the particles that have to be sent. Thus, using a similar approach as the one we used for `locateParticle` to limit its search to a subset of the MPI ranks could improve the subroutine.

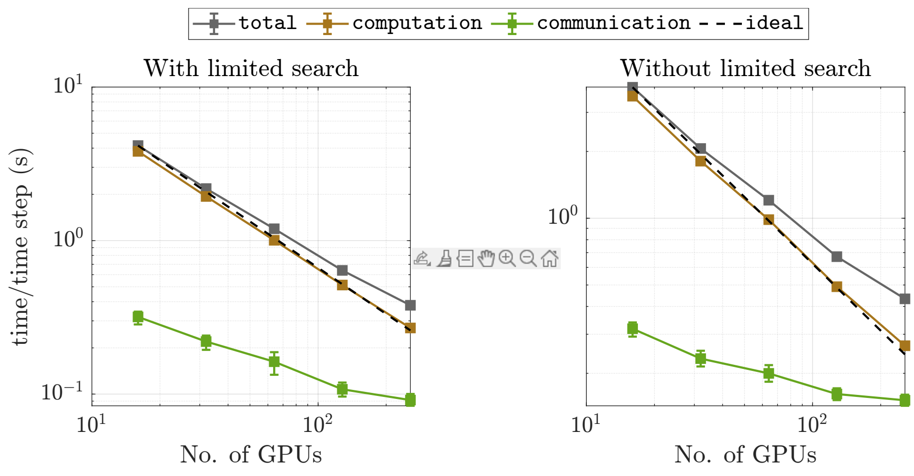


(a) Results for cases A (left) and C (right)

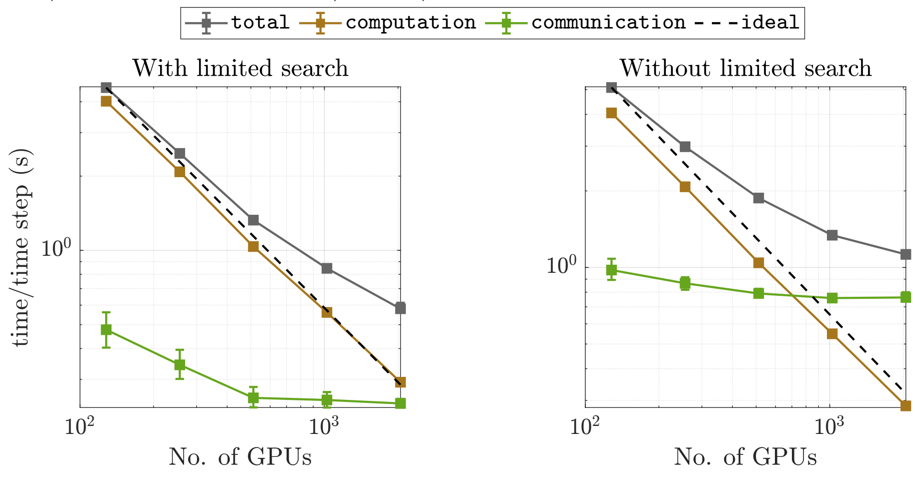


(b) Results for cases B (left) and D (right)

Figure 10: Runtimes on GPU of the functions `locateParticle` and `update` (referred as `updateParticle` in the legend). The curve `total` refers to the total simulation runtime.

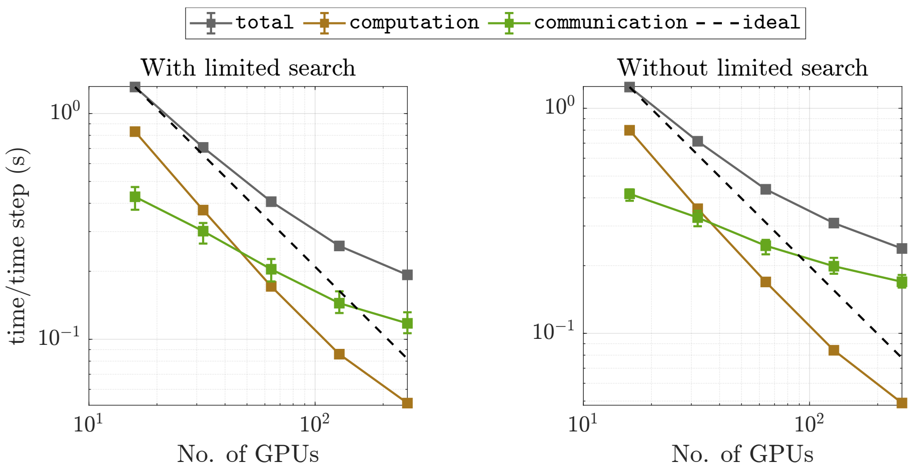


(a) Results for the new implementation (on the right) vs. the old one (on the left), with grid size 512^3 (case A).

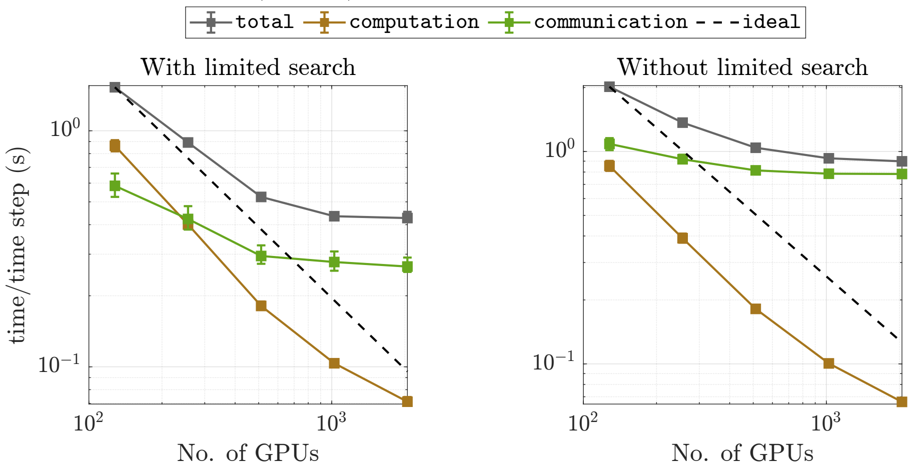


(b) Results for the new implementation (on the right) vs. the old one (on the left), with grid size 1024^3 (case B).

Figure 11: Global time spent on communication and computation for the GPU runs and 8 particles per cell (case A and B). In the graph, the total runtime is also shown, along with the ideal scaling.



(a) Results for the new implementation (on the left) vs. the old one (on the right), with grid size 256^3 (case C).



(b) Results for the new implementation (on the left) vs. the old one (on the right), with grid size 512^3 (case D).

Figure 12: Global time spent on communication and computation for the GPU runs and 64 particles per cell (cases C and D). In the graph, the total runtime is also shown, along with the ideal scaling.

```
using inner_t = Kokkos::View<int*>;

Kokkos::View<inner_t, Kokkos::SharedSpace>
    kArray("Array",my_std2dArray.size());

for(int i=0; i<my_KokkosArray.extent(0); i++) {
  Kokkos::View<int*, Kokkos::HostSpace>
    host_i(my_std2dArray[i].data(), my_std2dArray[i].size());
  kArray(i) = inner_t("Inner",host_i.extent(0));
  Kokkos::deep_copy(kArray(i), host_i);
}
```

Algorithm 4: Example of `SharedSpace` used to move a 2-D array in a View of Views.

References

[1] IPPL GitLab repository. https://gitlab.psi.ch/OPAL/Libraries/ippl.

[2] IPPL GitLab wiki. https://gitlab.psi.ch/OPAL/Libraries/ippl/-/wikis/home.

[3] Kokkos programming guide. https://kokkos.github.io/kokkos-core-wiki/programmingguide.html.

[4] OPAL GitLab wiki. https://gitlab.psi.ch/OPAL/src/-/wik/home.

[5] Andreas Adelmann, Pedro Calvo, Matthias Frey, Achim Gsell, Uldis Locans, Christof Metzger-Kraus, Nicole Neveu, Chris Rogers, Steve Russell, Suzanne Sheehy, Jochem Snuverink, and Daniel Winklehner. Opal a versatile tool for charged particle accelerator simulations, 2019.

[6] CSCS. Factsheet: “Piz Daint”, one of the most powerful supercomputers in the world. https://www.cscs.ch/publications/news/2017/factsheetpizdaintoneofthemostpowerfulsupercomputersintheworld/, 2017.

[7] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*,

74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[8] Michael Ligotino. Implementation of a load balancing scheme and domain decomposition in the independent parallel particle layer library. amas.web.psi.ch/people/aadelmann/ETH-Accel-Lecture-1/projectscompleted/cse/ORBMichael.pdf.

[9] Sriramkrishnan Muralikrishnan, Matthias Frey, Alessandro Vinciguerra, Michael Ligotino, Antoine J. Cerfon, Miroslav Stoyanov, Rahulkumar Gayatri, and Andreas Adelmann. Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures. https://arxiv.org/abs/2205.11052, 2022.

[10] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022.

Declaration of Originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of Master's thesis (in block letters):

IMPROVING PARTICLE COMMUNICATION IN THE INDEPENDENT PARALLEL PARTICLE LAYER

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
MONTANARO	VERONICA

With my signature I confirm that
− I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
− I have documented all methods, data and processes truthfully.
− I have not manipulated any data.
− I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZUERICH, 07/02/2023	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.