ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PSI

Integration and Testing of the P^{3}M Algorithm in IPPL

Semester Project

in Computational Science and Engineering

Department of Mathematics

ETH Zurich

written by

Jonas Bachmann

supervised by

Dr. Andreas Adelmann (ETH/PSI)

scientific adviser

Alecxnder Liemen (ETH)

February 2, 2026

Contents

Chapter 1

Introduction

The Particle-Particle-Particle-Mesh (P^3M) method is a computational technique used in N-body simulations [1–3] and molecular dynamics [4–6] to efficiently calculate (pairwise) long-range interactions, particularly electrostatic or gravitational forces. This hybrid approach addresses the computational challenge of calculating interactions between all particle pairs, which scales as $\mathcal{O}(N^2)$ and becomes infeasible for large systems.

The P^3M method builds upon and extends the fundamental Particle-in-Cell (PIC) method by combining PIC's mesh-based techniques with direct particle-particle calculations. While PIC methods typically handle all interactions through mesh interpolation and field solving, the P^3M method directly evaluates particle-particle interactions when particles are close together, resulting in more accurate description of the interactions. The P^3M method works by splitting interactions into short-range and long-range components. The long-range part (PM) of P^3M essentially employs the same core PIC methodology, interpolating particle properties to grid points, solving field equations in Fourier space leveraging fast Fourier transforms (FFTs) [7], and interpolating forces back to particle positions. Short-range interactions on the other hand are only evaluated between nearby particles using traditional particle-particle (PP) methods.

By splitting the interactions, the computational complexity changes as follows:

$$\mathcal{O}(N^2) \to \mathcal{O}(N \cdot n) + \mathcal{O}(M \log M),$$

where n is the average number of neighbor particles considered for each particle in the short-range interaction, and M is the total number of mesh points and thus governs the runtime of the FFTs. The computational complexity of P^3M allows scaling to larger systems, assuming n and M are kept small enough.

This hybridization allows the P^3M method to maintain the computational efficiency gains of PIC methods for long-range effect while achieving higher accuracy for short-range interactions through direct calculation, representing an optimal balance between the $\mathcal{O}(N^2)$ scaling of pure particle-particle methods and the potential accuracy limitations of pure mesh-based approaches.

1.1 Previous Work

This project builds on the previous works of Ulmer [8] and Schwab [9]. Ulmer has implemented the P^3M method in OPAL (Object Oriented Parallel Accelerator Library), a parallel open source tool for charged-particle optics in linear accelerators and rings [10]. It is written in C++ for CPU with MPI to harness the power of parallel processing for large scale machines.

The developers of OPAL have recently decided to modernize the code base to the C++20 standard and rebranding as OPALX [11]. Besides modernizing the code, another big reason to rewrite OPAL is adding GPU support. Central to this modernization effort is IPPL (Independent Parallel Particle Layer) [12, 13], the core computational framework that handles particle and field operations within OPAL. In the revision process, IPPL utilizes Kokkos [14, 15] to achieve performance portability across various hardware architectures.

Schwab has reproduced Ulmers results, specifically the disorder-induced heating (DIH) problem [2], using the newer version of IPPL and Kokkos. This project integrates Schwabs DIH example and the corresponding methods into IPPL.

1.2 Project Overview

This project aims to incorporate the P^3M method into IPPL. To this end the short-range part will be generalized to D dimensions. Furthermore, we need a consistent particle neighbor list, a new concept; particle interaction solvers (the short-range pendant to field solvers), and proper boundary conditions. A central focus of this project are particle pairs across domain decomposition boundaries. The long-range part is already a part of IPPL.

Chapter 2

Background

2.1 The P^{3}M Method

The derivation of the P^{3}M method given here closely follows the original algorithm presented by Hockney and Eastwood [1].

2.1.1 Particle-Particle Interactions

We study a system of N discrete point-like particles at positions $\mathbf{x}_i \in \mathbb{R}^3$ with charges $q_i \in \mathbb{R}$. In the following derivation we write

$$\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i, \quad r_{ij} = \|\mathbf{r}_{ij}\|, \quad \text{and} \quad \hat{\mathbf{r}}_{ij} = \frac{\mathbf{r}_{ij}}{r_{ij}},$$

for the distance vector, the norm of the distance, and the unit vector between two particles i and j.

Coulomb's Law. The electrostatic force between two charged particles i and j is given by Coulomb's law as

$$\mathbf{F}_{ij} = k_e \frac{q_i q_j}{r_{ij}^2} \hat{\mathbf{r}}_{ij}, \tag{2.1}$$

where $k_e = \frac{1}{4\pi\epsilon_0}$ is Coulomb's constant and ϵ_0 the electric permittivity. Thus the force felt by a given particle i in the system is

$$\mathbf{F}_i = k_e q_i \sum_{i \neq j} \frac{q_j}{r_{ij}^2} \hat{\mathbf{r}}_{ij}. \tag{2.2}$$

The Coulomb potential for a single particle is given by

$$\Phi_i = k_e \sum_{i \neq j} \frac{q_j}{r_{ij}}. \tag{2.3}$$

With the corresponding potential energy $U_i = q_i \Phi_i$. One can easily verify $-\nabla_{\mathbf{r}_i} U_i = \mathbf{F}_i$. Finally the electrostatic field felt by particle i is given by

$$\mathbf{E}_i = -\nabla_{\mathbf{r}_i} \Phi_i = \frac{\mathbf{F}_i}{q_i}. \tag{2.4}$$

Poisson's Equation. The electric potential Φ and the charge distribution ρ are as

$$\nabla^2\Phi = -\frac{\rho}{\epsilon_0},$$

which is called the Poisson's equation. The electric potential is given by eq. (2.3). In the continuous limit it can instead be written as

$$\Phi(\mathbf{x}) = k_e \int \frac{\rho(\mathbf{x}')}{\|\mathbf{x}-\mathbf{x}'\|}\mathrm{d}\mathbf{x}',$$

which can be interpreted as the Green's function solution of Poisson's equation, with Green's function $G(\mathbf{r}) = \frac{1}{\|\mathbf{r}\|}$. Thus the the integral can be rewritten as

$$\begin{aligned}\Phi(\mathbf{x}) &= k_e \int G(\mathbf{x}-\mathbf{x}')\rho(\mathbf{x}')\mathrm{d}\mathbf{x}' \\ &= k_e(G * \rho)(\mathbf{x}),\end{aligned}$$

where the asterisk denotes a convolution. In Fourier space a convolution is a simple multiplication and can be solved efficiently with FFTs. Thus we can write in Fourier space

$$\hat{\Phi}(\boldsymbol{\xi}) = k_e\hat{G}(\boldsymbol{\xi})\hat{\rho}(\boldsymbol{\xi}), \quad \boldsymbol{\xi} \in \mathbb{R}^3$$

where the hat denotes the Fourier transform of the respective functions.

2.1.2 Hockney and Eastwood P^{3}M Algorithm

Hockney and Eastwood have presented a hybrid algorithm to efficiently account for long-range and short-range effects between charged particles in [1, chapter 8]. The central idea is to split the Coulomb force eq. (2.1) in short-range and long-range contributions as

$$\mathbf{F}(r) = \mathbf{F}^{\mathrm{SR}}(r) + \mathbf{F}^{\mathrm{LR}}(r). \tag{2.5}$$

The **long-range forces** can be computed by solving Poisson's equation in Fourier space using a grid based method such as particle in cell (PIC) method [16]. This changes the computational cost to compute all pairwise interactions from $\mathcal{O}(N^2)$ to $\mathcal{O}(M \log M)$, where M is the number of grid points of the mesh used for the FFT. Ensuring $M \lesssim N$ results in a computational speedup and allows the treatment of larger systems.

The **short-range interactions** are directly evaluated in real space as in eq. (2.2). However only particle pairs within a certain cutoff radius r_c are considered. The computational cost to compute all short-range interactions is thus $\mathcal{O}(N \cdot n)$, with n the average number of particles within a distance of r_c to each particle. Ulmer [8] studied the influence of the cutoff radius r_c on the balance between computational efficiency and accuracy. A suitable data structure is necessary for efficient computation and iteration of particle neighbors. Hockney and Eastwood suggest the use of a cell-list structure, however there are many other options such as Verlet-list [17], cell-linked list [18], tree-based structures (on GPU) [19] and more [20]. In this project the chain-list approach from Hockney and Eastwood is used, however future work should investigate other methods as well.

The P^{3}M method consists of two parts, the particle-particle (PP) and particle-mesh (PM) steps. We assume that we are given a uniform cartesian mesh $\mathcal{M}_{\mathrm{PM}}$ for the PM computation, thus the algorithm presented in [1] consists of three major steps and read as follows:

1. **PM force calculations**: An order p extrapolation scheme is used to deposit the charges q_i onto the mesh $\mathcal{M}_{\mathrm{PM}}$. Then the discrete long-range potential can be computed on the mesh using FFTs to solve Poisson's equation. The potential is interpolated back to the particles according to eq. (2.4) using an order p interpolation scheme.
2. **PP force calculations**: For every particle i search all neighboring particles within a cutoff radius r_c and compute the short-range force acting on it $\mathbf{F}_i^{\mathrm{SR}}$.
3. **Integration of Equation of Motions**: Update position and momenta under desired boundary conditions according to some time stepping method (e.g. Runge-Kutta methods) using the total forces computed for every particle $\mathbf{F}_i$. We use the second order symplectic Leapfrog scheme.

Repeat this steps until the total desired simulation time is reached.

2.1.3 Interaction Force Splitting

Instead of splitting the Coulomb force eq. (2.1) to obtain eq. (2.5), we can instead split Green's function into a short-range ψ and long-range ϕ contribution as

$$G(r) = \frac{1}{r} = \psi(r) + \phi(r). \tag{2.6}$$

The splitting of the Coulomb force can then be derived from the splitting of Green's function. A meaningful splitting requires that short-range term $\psi(r)$ smoothly goes towards zero at the cutoff r_c such that the computation of pairwise interaction can be truncated at this cutoff. While the long-range term $\phi(r)$ should be smooth and non-singular. Advantages and disadvantages of different type of splitting schemes are discussed in [21, 22], we choose Gaussian shaped charges with the splitting reading

$$G(r) = \psi(r) + \phi(r) = \frac{1 - \mathrm{erf}(\alpha r)}{r} + \frac{\mathrm{erf}(\alpha r)}{r},$$

with $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t$ the error function and α a parameter to control the range of the short-range term. Motivated by Ulmer's results [8], for the rest of the project we set

$$\alpha = \frac{2}{r_c}.$$

The splitting of Green's function into short-range and long-range parts is shown in fig. 2.1 with $\alpha = 2$.

Long-Range Interaction

The treatment of the long-range term $\phi(r)$ in the PM step and how to compute its Fourier transform has been done extensively by Ulmer [8]. As we will not change the PM step it is not further discussed here.

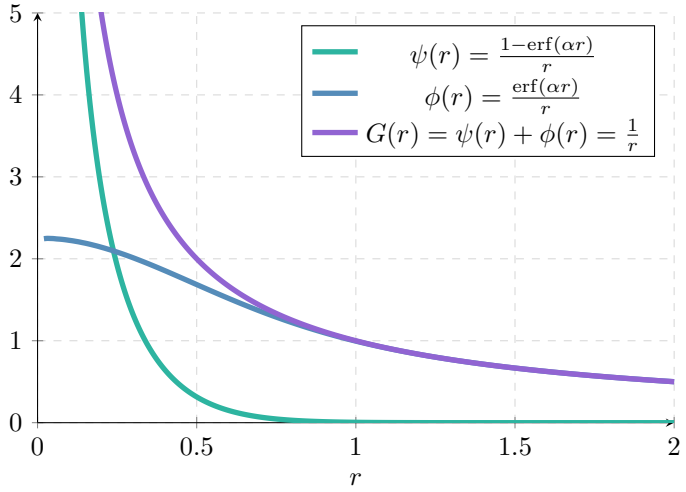


Figure 2.1: Green's function splitting based on Gaussian shaped charges with $\alpha = 2$.

Short-Range Interaction

The explicit form of the short-range force can now be computed according to eq. (2.4) as

$$\begin{aligned}\mathbf{F}_i &= -q_i \nabla_{\mathbf{r}_i} \Phi_i, \\ &= -q_i \nabla_{\mathbf{r}_i} \left[k_e \sum_{j \in \mathcal{N}(i)} q_j \psi(r_{ij}) \right], \\ &= -k_e q_i \sum_{j \in \mathcal{N}(i)} q_j \nabla_{\mathbf{r}_i} \psi(r_{ij}),\end{aligned}$$

with $\mathcal{N}(i)$ the particle neighbor indices of particle i. The gradient of the short-range term is

$$\begin{aligned}\nabla_{\mathbf{r}_i} \psi(r_{ij}) &= -\hat{\mathbf{r}}_{ij} \nabla_{r_{ij}} \psi(r_{ij}), \\ &= -\hat{\mathbf{r}}_{ij} \nabla_{r_{ij}} \left[\frac{1 - \mathrm{erf}(\alpha r_{ij})}{r_{ij}} \right], \\ &= -\hat{\mathbf{r}}_{ij} \left[\frac{2\alpha e^{-\alpha^2 r_{ij}^2}}{\sqrt{\pi} r_{ij}} + \frac{1 - \mathrm{erf}(\alpha r_{ij})}{r_{ij}^2} \right].\end{aligned}$$

Thus the short-range part of the Coulomb force eq. (2.1) can be written as

$$\mathbf{F}_{ij} = k_e q_i q_j \left[\frac{2\alpha e^{-\alpha^2 r_{ij}^2}}{\sqrt{\pi} r_{ij}} + \frac{1 - \mathrm{erf}(\alpha r_{ij})}{r_{ij}^2} \right] \hat{\mathbf{r}}_{ij}.$$

2.1.4 Application to Different Domains

Even though we have exclusively talked about charged particles, the derivation above and with it the P^3M method can be applied to any pairwise force of the same functional form as the Coulomb force, $\mathbf{F} \propto \frac{1}{r^2}$. An example would be the gravitational force between two planets with masses m_1, m_2 separated by distance r

$$F = G\frac{m_1 m_2}{r^2},$$

with G the gravitational constant.

2.2 Kokkos

Kokkos is a C++ performance portability framework [14, 15] designed to enable efficient parallel computing across diverse hardware architectures, including CPUs, GPUs, and other accelerators. Kokkos provides a unified programming model that abstracts hardware-specific details while maintaining performance across different execution environments. While Kokkos provides portability for most node configurations used in today's HPC clusters, parallelization across multiple processes requires MPI for message passing between the different processes.

2.2.1 Device-Host Separation

One of Kokkos' fundamental design principles is the clear separation between host and device memory and execution spaces. This abstraction allows developers to write portable code that can execute on different hardware without modification. The framework manages memory allocation and data movement between host (CPU) and device memory spaces transparently. Host allocated memory can be used in host execution space, while device allocated memory can be used in the device execution space. All functions and data-structures in Kokkos are templated on the execution space and/or memory space respectively. There are two key type definitions:

- `Kokkos::DefaultHostExecutionSpace` is a type alias of the highest ExecutionSpace available in the hierarchy `host-parallel`, `host-serial`.
- `Kokkos::DefaultExecutionSpace` is a type alias of the highest ExecutionSpace available in the hierarchy `device`, `host-parallel`, `host-serial`.

The spaces which are available depends on what (Kokkos) configuration the code is compiled for. If needed, spaces can be provided via template arguments more specifically.

2.2.2 Kokkos Views

Kokkos Views are the primary data structure for managing multi-dimensional arrays with automatic memory management across different memory spaces. Views provide a unified interface for accessing data regardless of the underlying memory layout or location. Views manage the memory and `Kokkos::deep_copy` is needed to actually create a copy of the underlying data.

Listing 2.1: Example construction and device to host memory transfers using Kokkos views.

```
// 1D view declaration
Kokkos::View<double*> vector("my_vector", N);

// 2D view with specific memory space
Kokkos::View<double**, Kokkos::Serial> matrix("my_matrix", rows, cols);

// Creating mirror views for host-device data transfer
auto host_vector = Kokkos::create_mirror_view(vector);
Kokkos::deep_copy(host_vector, vector);  // Device to host
Kokkos::deep_copy(vector, host_vector);  // Host to device
```

By default views are allocated in `Kokkos::DefaultExecutionSpace::memory_space` but the memory space can be specified explicitly. Two important details to be pointed out

1. Entries of a view (`vector(i)`) can only be accessed from the corresponding execution space. Thus because of point 1. its possible to index into a device view in the host execution space when compiling for e.g. OpenMP, but this will segfault when compiling the same code with e.g. CUDA as the device.

2. If the device has the same memory space as the host (i.e. the device is CPU based) device to host mirroring (`Kokkos::create_mirror_view(...)`) and vice-versa will give you a view pointing to the same underlying memory, as device and host memory space are the same. `Kokkos::create_mirror(...)` on the other hand will always provide you with newly allocated memory.

2.2.3 Parallel Execution Patterns

Kokkos provides several parallel execution patterns that abstract common computational kernels. The most fundamentals are patterns are summarized below.

- `Kokkos::parallel_for` executes a Kokkos lambda function or functor in parallel across a specified range or index space.
- `Kokkos::parallel_reduce` reduces value in parallel across a specified range or index space. Common reductions are addition, multiplication or min/max. The functor is called exactly once for each index.
- `Kokkos::parallel_scan` computes the exclusive or inclusive scan in parallel across a specified range or index space. Note that the functor might be called multiple times for some indices.

Listing 2.2: Example usage of Kokkos' parallel execution patterns.

```
// Simple parallel loop over a range
int N = 1000000;
double a = 3;
Kokkos::parallel_for("axpy", N, KOKKOS_LAMBDA(int i) {
    y(i) = y(i) + a*x(i);
});

// Multi-dimensional parallel reduction
double frobenius_norm = 0;
Kokkos::parallel_reduce("frobenius_norm",
    Kokkos::MDRangePolicy<Kokkos::Rank<2>>({0, 0}, {N, N}),
    KOKKOS_LAMBDA(int i, int j, double& sum) {
        sum += a(i, j) * a(i, j);
    },
    Kokkos::Sum<double>(frobenius_norm)
);
```

To ensure the functors passed to the parallel execution patterns can be compiled for the various hardwares, lambdas need to be defined with the macro `KOKKOS_LAMBDA` as in listing 2.2 and functors need to have their call operator annotated with `KOKKOS_FUNCTION` or a variant of it. These macros are expanded differently for different targets and make sure that e.g. when

compiling for CUDA, the lambdas and functions are annotated with `__host__ __device__` to make sure compilation is done both for the host and the device code.

The combination of these features enables Kokkos to deliver performance portability across diverse hardware architectures. The same source code can be compiled and executed efficiently on multi-core CPUs, NVIDIA GPUs (via CUDA), AMD GPUs (via HIP), and Intel GPUs (via SYCL) by simply changing the execution space at compile time. This abstraction significantly reduces code duplication and maintenance overhead in performance portable computing applications.

2.3 IPPL

The development of the Independent Parallel Particle Layer (IPPL) C++ Library [12, 13] started about 20 years ago. The general framework is designed to develop Lagrangian, Eulerian, and hybrid Eulerian-Lagrangian solvers. IPPL uses MPI to distribute particles and fields accross multiple processes to achieve massive parallelization on modern supercomputers. IPPL has recently been revised to modern C++20 standard and has adopted the Kokkos programming model to achieve performance portability across various hardware architectures [13]. The following overview closely follows [13, section 2].

Figure 2.2 shows a schematic overview of how IPPL treats particles. For cache aware iteration of the particles, IPPL uses the paradigm *struct of arrays*, where each attribute (`ParticleAttrib`) is a Kokkos View enhanced with expression templates. An attribute can be anything which is defined per-particle such as position, mass and charge or even a global index. Note that the underlying type of an attribute is not restricted to scalar types only, vector type attribute such as position and velocity are thus *arrays of structs*. Every particle collection will per default have position (`R`) and optionally globally unique indices (`ID`) as attributes. The class `ParticleLayout` manages the distribution of the particles among processes. Users must define their own particles class deriving from `ParticleBase`. Additional user defined particles attributes can easily be added with `addAttribute`. This makes `ParticleBase` aware of the new attributes and makes sure they are included in MPI exchange operations. To send and receive particles from one process to another, all particle attributes of the exchanged particles are serialized to one linear buffer by the sending process and then de-serialized by the receiving process. An example of how to define a particle container is shown in listing 2.3.

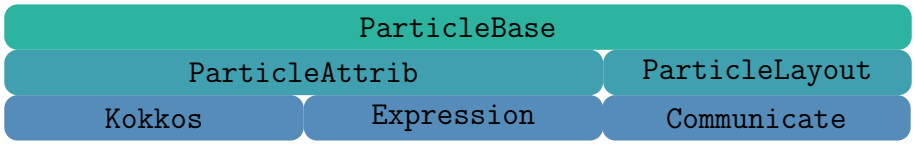


Figure 2.2: General layout of the particle container class (`ParticleBase`) in IPPL. The underlying container of each particle attribute (`ParticleAttrib`) is a Kokkos View. The communication among processes is managed by its layout (`ParticleLayout`).

For a detailed description of IPPL's treatment of fields (needed for the PM step in the P^3M method) we refer to [13, section 2].

Listing 2.3: Example of a particle container class with additional particle attributes (mass and velocity). The design of IPPL allows the user to easily add new attributes. Mathematical expressions compile to a single Kokkos kernel without storing any temporaries, thanks to the use of C++ expression templates.

```
using namespace ippl;
template <class PLayout>
struct Particles : public ParticleBase<PLayout> {
    Particles(PLayout& playout) : ParticleBase<PLayout>(playout) {
        // add additional attributes to ParticleBase
        this->addAttribute(mass);
        this->addAttribute(velocity);
    }
    ParticleAttrib<double> mass;
    ParticleAttrib<Vector<double>> velocity;
};

// compiles to single Kokkos kernel
particles->R = particles->R + dt * particles->velocity;
```

2.4 Existing Implementation

As discussed in section 1.1 Schwab has implemented the P^3M method using the IPPL framework [9]. The goal is to adapt his code to the philosophy and code style of IPPL and extend it to be more general. Here we discuss what parts of Schwab's work are ready to merge into IPPL and what parts need some more work.

2.4.1 PM Step

There are various Poisson solvers in IPPL such as conjugate gradient, periodic FFT, open FFT, and FEM solvers. And there is also a solver to solve the Poisson equation with the truncated Green's function as derived in eq. (2.6). Previously this solver was called `P3MSolver`, however as it only solves the long-range part of the P^3M method, to be pedantic it was renamed to `FFTTruncatedGreenPeriodicPoissonSolver`. As IPPL was built with particle-in-cell methods in mind and the long-range part part of the P^3M method follows the same methodology as PIC, no further modifications are required for the PM step.

2.4.2 PP Step

In particle-in-cell methods particle are advected using the forces obtained from computations on the field. The P^3M method additionally requires particle-particle interactions and to that end a way to iterate over particle pairs.

The following properties are all from Schwab's implementation [9].

- **Particle Neighbor List** The particle neighbor list currently only works for 3 dimensions. Pairs of particles across multiple processes are done via halo exchange of the underlying cells of the particle neighbor list by hardcoding the exchange to and from all 27 neighbors in 3 dimensions.

- **Boundary Conditions** Periodic boundary conditions are not possible as they introduce new interaction pairs over the domain boundary.
- **Particle Interaction Solver** The particle-particle interactions are computed by iterating through all pairs in the particle neighbor list and halo exchanging the boundary particles. This is again done for 3 dimensions only.

The goal of this project is to generalize the particle neighbor list to D dimensions, incorporate the halo exchange into existing MPI calls and introduce a new type of solvers for particle-particle interactions, similar to the Poisson solvers in IPPL.

Chapter 3

Implementation in IPPL

This chapter gives detailed insight over the implementation of the P^{3}M method within the IPPL library. The main contribution of this project is the creation of a new particle layout and a new solver type for short-range interactions. The IPPL library containing all the implementations discussed in this chapter can be found on GitHub via the link github.com/IPPL-framework/ippl.

3.1 New Particle Layout

The particle layout (`ParticleLayout` in fig. 2.2) determines how all particles are distributed among the processes. A particle layout needs to provide a function `update(ParticleContainer& pc)` which determines which process stores each particle and exchanges the particles accordingly.

A new particle layout is introduced `ParticleSpatialOverlapLayout`. It aims to solve the limitations discussed in section 2.4.2, namely generalizing the particle neighbor list to D dimensions, including exchange of inter process particle pairs in existing MPI routines, and adding periodic boundary conditions. It inherits from `ParticleSpatialLayout`, but additionally has a particle neighbor list. A schematic view is given in fig. 3.1.

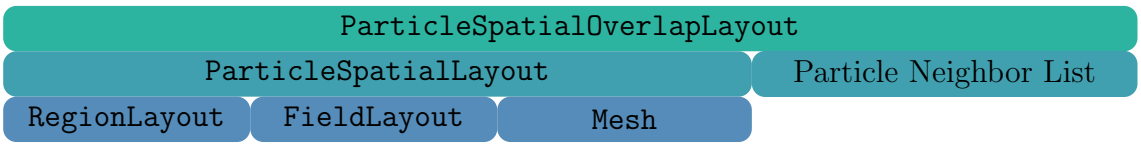


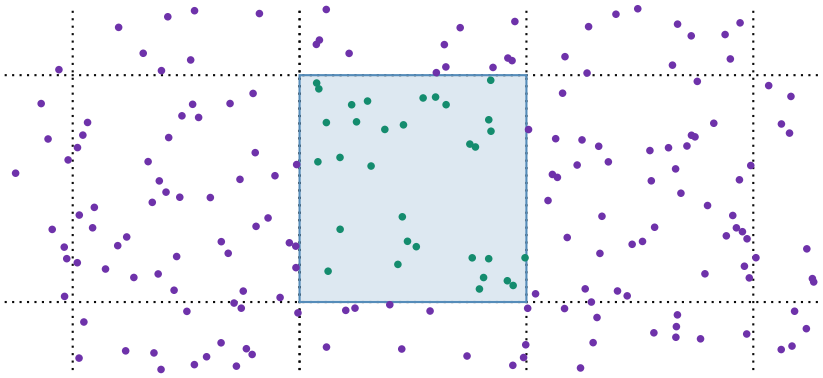
Figure 3.1: Layout of the class `ParticleSpatialOverlapLayout` in IPPL. It is an extension of the class `ParticleSpatialLayout`, with the addition of a particle neighbor list. The domain decomposition is handled by `RegionLayout`, while the information of process neighbors is managed by `FieldLayout`. The particle neigbor list is not a class itself but an integral part of `ParticleSpatialOverlapLayout`

Just like `ParticleSpatialLayout` the overlap layout distributes particles according to their position but as the name suggests, there is some overlap, i.e. some particles can be present in multiple processes at the same time. Only one copy of every particle is considered the *real* one, the rest of its copies are *ghost* particles. A comparison between the two layouts can be seen in fig. 3.2. A few questions immediately arise:

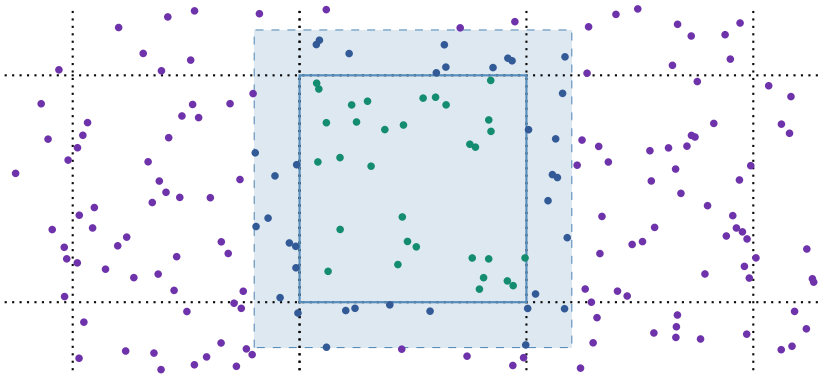
1. How to store the *ghost* particles?

2. How to build a particle neighbor list?
3. How to implement the domain overlap?
4. How to exchange particles now that they can be in multiple domains?
5. How to deal with particles at the global domain boundary in case of periodic boundary conditions?

The first two questions are answered in section 3.1.3, the third and fourth one are answered in section 3.1.2 and the fifth one is answered in section 3.1.6.



(a) Schematic of `ParticleSpatialLayout` layout from the view of the blue domain (solid line). The blue domain stores the green particles. Purple particles are not stored by the blue domain.



(b) Schematic of `ParticleSpatialOverlapLayout` from the view of the blue domain (solid line) and its overlap (dashed line). The blue domain stores the green particles (*interior* particles) and the blue particles (*ghost* particles). The blue particle and some of the green particles close to the boundary are also stored by other domains. Purple particles are not stored by the blue domain.

Figure 3.2: Comparison between where particles are stored in `ParticleSpatialLayout` (a) and in `ParticleSpatialOverlapLayout` (b).

3.1.1 Offset Arrays

Here we discuss how to deal with inhomogeneous data in parallel. Lets say we have N buckets and M labeled balls and we want to know the distribution of balls in the buckets, i.e. we want to know which balls are in which bucket. In serial code one may use an inhomogenous two

dimensional structure. A mental model can be something like a vector of (dynamic) vectors like `std::vector<std::vector<...>>` (even though this is not very cache friendly). This kind of problem is a common pattern and is required several times throughout this project and can be solved more efficiently by offset arrays. It is a technique similar to how some sparse matrix storage formats (e.g. CRS) work. The idea is to store all data in a (linear) contiguous array (call it `balls`) and have a second array used to index into data (call it `bucket_offsets`). The offset array data structure can be computed in a three pass algorithm

1. Compute how many entries there are for every index, i.e. collect the number of balls in each bucket and sore it in an array, call it `bucket_sizes`.
2. Compute the the exclusive scan of `bucket_sizes` to obtain `bucket_offsets`. For consistency set `bucket_offsets(N) = M`. Now `bucket_sizes` is not needed anymore.
3. Fill `balls` with the labels of the balls according to which bucket they belong to.

Note that `balls` has size `M` and `bucket_offsets` size `N+1`. Thus finally bucket `i` contains the balls `balls(bucket_offsets(i)), ..., balls(bucket_offsets(i + 1) - 1)`. All three steps can be done easily with the help of Kokkos' parallel execution patterns.

3.1.2 Particle Exchange

As the first step in `ParticleSpatialOverlapLayout::update(...)` the particles are redistributed as illustrated in fig. 3.2(b). The second step is then to build a particle neighbor list, this is discussed in section 3.1.3.

In contrast to `ParticleSpatialLayout` in the overlap layout each particle can be present on multiple processes. This complicates the algorithm to redistribute the particles slightly but still follows the four steps from `ParticleSpatialLayout` to perform the MPI based particle exchange

1. Figure out which particles need to go to which process.
2. Fill send buffer and send particles.
3. Delete invalidated particles.
4. Receive particles.

The first step needs to be changed the most to respect the domain overlaps. If we talk about comparisons in the following algorithm description, we always refer to the contrast between `ParticleSpatialLayout` and the new `ParticleSpatialOverlapLayout`.

Locating Particles

Instead of having a single view storing the process id each particle belongs to, we need information about all the processes each particle belongs to. To this end offset arrays (see section 3.1.1) are employed. To avoid checking all processes for all particles, first only the process which currently owns the particle and its $3^D - 1$ nearest neighbor processes are checked, with D the dimension. We make the assumption, that the overlap cannot be bigger than half the domain size of any domain. With this assumption if a given particle also belongs to the currently owning process the particle can at most be in it's nearest neighbor processes, and thus we do not need to chack any other processes. This is the case as the furthest a particle can be away from it's currently owning process is half the smallest domain size in any dimension and thus it cannot belong to any second neighbor.

To keep the same domain class `RegionLayout` (see fig. 3.1) the overlap is simply implemented by an additional scalar which can be added to the domain maximum and subtracted from the

domain minimum when checking a particle's position. Thus to check whether a particle with position `pos` belongs to a certain `region` ane can use the following boolean condition:

```
(pos[0] > region[0].min() - overlap)
    && (pos[0] <= region[0].max() + overlap) && ...
```

This can be expanded automatically for any dimension with C++ template parameter packs and fold operations [23].

Pack and Send Particles

To send the particles to a given process, the indices of all particles which need to be sent to this process need to be collected. Now instead of just checking whether a given particle belongs to the given process, we need to check all processes of the particle and see whether the given process is among them.

Delete Invalid Particles and Receive Particles

Works the same as in `ParticleSpatialLayout`.

Now each process contains their *real* and *ghost* particles `particleContainer.getLocalNum()` will return the sum of both. This will be adjusted at the end of the construction of the particle neighbor list.

3.1.3 Particle Neighbor List

The domain overlap allows for a consistent particle neighbor list, as there is no problem with particles close to a domain boundary. The particle neighbor list implemented by Schwab and extended by this project is of cell list type. The particle neighbor list works by only considering particle pairs from the same cell or neighboring cells. The cell list can be built as follows and is illustrated in fig. 3.3:

1. Create a tensor-product grid from the domain (including overlap region).
2. Bin all particles into their respective cell depending on their position.
3. Store the particles according to the index of the cell they are in.

How to store the the particles? We want to keep the architecture of IPPL, storing particle in classes derived from `ParticleBase`. Thus we will not change the location where particle's attributes are stored but we sort them in a smart way allowing us to efficiently iterate over all particle pairs. The particles are sorted in a way which keeps all particles of the same cell in a contiguous index range. This only requires us to keep track of which cell starts at which particle index. Remember that the attributes of the particles are stored in one dimensional views but the cells are a D dimensional grid. Thus a index function is required to map from the D dimensional cell index to the flat one dimensional index. To bin the particles into the cells the D dimensional index can be computed as shown in listing 3.1. From this there is a natural way how to compute the flat one dimensional index:

```
flatIndex = cellIndex[0] + numCells[0] * cellIndex[1]
                         + numCells[0] * numCells[1] * cellIndex[2] + ...
```

with `numCells` the number of cells in each dimension. Choosing one layer of *ghost* cells for the overlap gives us an immediate connection between the overlap amount and the cutoff radius r_c

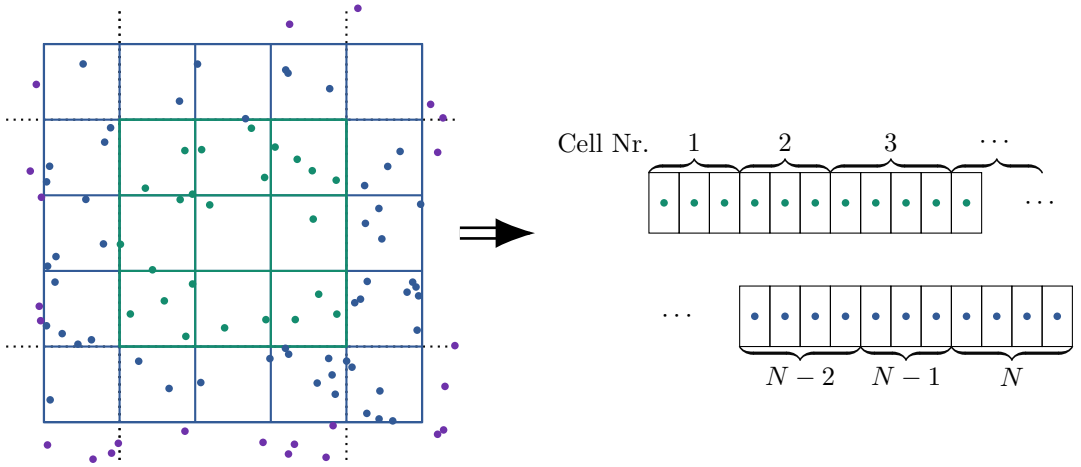


Figure 3.3: Cell list structure of the particle neighbor list with N total cells (including *ghost* cells). Green particles are *real* particles of the given domain and are stored at the beginning of the attribute views. The blue particles are *ghost* particles and are stored at the end of the attribute views. One layer of cells is used for the overlap.

Listing 3.1: Computation of the D dimensional cell index for a particle at position `pos`. The local region is stored in `region`. Note the `+1` at the end, this is because the the ghost particles lay outside the local region (at most one cell) and otherwise the indices would be negative.

```
for (unsigned d = 0; d < Dim; ++d) {
    cellIndex[d] = Kokkos::floor((pos[d] - region[d].min()) /
    ↪  cellWidth[d]) + 1;
}
```

Listing 3.2: Computation of the cell width from the cutoff radius which is equal to the the overlap. This ensures the cell width is at least as big as r_c in each dimension.

```
for (unsigned d = 0; d < Dim; ++d) {
    const auto length = region[d].length();
    const auto numCells = Kokkos::floor(length / rcutoff_m);
    cellWidth_m[d] = length / numCells;
}
```

from section 2.1.2. The overlap should be at least as big as r_c. To ensure a whole number of cells the width of the cells is computed as in listing 3.2.

With this computations we cen give an upper limit of of the distance between two particles.

$$r \leq \sqrt{h_x^2 + h_y^2 + h_z^2 + \ldots},$$

width h_i the cell width in a given dimension. Assuming the cell widths are close to the cutoff radius (which they are given the domain extent is sufficiently larger to the cutoff) we get

$$r \lesssim \sqrt{D} r_c,$$

with D the dimension. And as the cell with in each dimension is at least as big as the cutoff radius, we ensure that no two particles within a distance of r_c are more than on cell apart.

Cell Permutation

Note that in fig. 3.3 it is mentioned that the *ghost* particles should be stored at the end of the attributes. So far the index computation would not respect this. But why is this important? As the number of particles on every process likely changes in every step as particles travel across domain boundaries, IPPL does not resize the particle attributes to the exact number of local particles every step but the views are treated as a buffer plus a size (the number of process local particles). Thus the memory of the attributes is only increased if more space is needed iterations over them should only be done depending on the number of process local particles. Coming back to the question; putting the *real* particles at the beginning allows us to tell the particle container that there are only as many process local particles, while the `ParticleSpatialOverlapLayout` knows about the additional *ghost* particles past the *real* particles. Note that like this no part of the code needs to be aware of the *ghost* particles besides the layout. To make sure this separation is done, an additional permutation of the one dimensional cell index can be introduced. These are `cellPermutationForward` and `cellPermutationBackward` such that:

```
flatIndex = cellPermutationForward[flatIndex]
```

The inverse permutation can be achieved with `cellPermutationBackward`. From now on we refer to cell `i` as the permuted cell index.

Particle Neighbor List Computation

Now we can discuss how to compute the cell list structure. First the particles are binned into the cells. Thus we count how many particles are in each cell (`cellParticleCount`). The atomics in listing 3.3 are unavoidable, as we have no knowledge of how the particles are distributed after the particle exchange (see section 3.1.2).

Listing 3.3: Parallel computation of `cellParticleCount`. Note that `numLoc` refers to all particles (*real* and *ghost*) at this moment.

```
Kokkos::parallel_for(numLoc, KOKKOS_LAMBDA(const size_t& i) {
    const auto flatCellIndex = getFlatCellIndex(positions(i), ...);
    Kokkos::atomic_increment(&cellParticleCount(flatCellIndex));
});
```

Next we need the indices of where each cell starts in the particle views. This view is be called `cellStartingIdx` and works just the same as `bucket_offsets` in section 3.1.1. Thus `cellStartingIdx` is the exclusive scan of `cellParticleCount`. Finally the particles need to be sorted such that `cellStartingIdx` points to the correct particles. This can be done with a second pass over all particles and keeping track of which ones have been encountered so far in each cell. To this end take a copy of `cellStartingIdx` (`cellCurrentIdx`) and assign particle `i` the new index `++cellCurrentIdx(i)`. For this increment again atomics are required as can be seen in listing 3.4.

Finally the particles need to be sorted according to their new indices `newIndex`. How the particles and all of its attributes are sorted is discussed in section 3.1.5. And the local number of particles in the particle container is now set to only the *real* particles, such that only the

Listing 3.4: Parallel computation of `newIndex`. Note that `numLoc` refers to all particles (*real* and *ghost*) at this moment.

```
Kokkos::deep_copy(cellCurrentIdx, cellStartingIdx);

Kokkos::parallel_for(numLoc, KOKKOS_LAMBDA(const size_type& i) {
    const auto flatCellIndex = getFlatCellIndex(positions(i), ...);
    newIndex(i) =
    ↪  Kokkos::atomic_fetch_inc(&cellCurrentIdx(flatCellIndex));
});
```

layout and the particle neighbor list know about the ghost particles. Like this all other routines concerning the particles only iterate over the *real* particles.

3.1.4 Iteration over Particle Pairs

We have implemented functions to get all neighbor particles of a given particle index or position. To use these functions in Kokkosi parallel execution patterns they should be `static KOKKOS_FUNCTION`'s to avoid copies of the layout class (done by `KOKKOS_CLASS_LAMBDA`). To achieve this we provide the function `getParticleNeighborData()` to get a proxy of the data required for the particle neighbor functions, see listing 3.5. Even though these functions are provided they are not the most efficient way to iterate over all particle pairs. As the layout knows

Listing 3.5: Functions to get particle neighbors of a given particle index (top) or position (bottom). Function to get proxy of the necessary particle neighbor list data is included.

```
ParticleNeighborData getParticleNeighborData() const;

particle_neighbor_list_type getParticleNeighbors(index_t particleIndex,
↪  const ParticleNeighborData& particleNeighborData);
particle_neighbor_list_type getParticleNeighbors(const vector_type&
↪  pos, const ParticleNeighborData& particleNeighborData);
```

all the implementation details of the particle neighbor list, its best if the layout itself provides a way to iterate over all pairs. This way any future layout with their own particle lis can decide how to iterate through the pairs. To this end we propose the following design choice: a function `forEachPair(...)` (see listing 3.8) taking a functor and templated on `ExecutionSpace`. The functor is called with the indices of the particle pairs and these indices are generated in the given `ExecutionSpace`.

Listing 3.6: Function provided by `ParticleSpatialOverlapLayout` to iterate over all particle pairs. The functor `f` will be called with indices `i` and `j` as `f(i, j)` where `i` are only indices of *real* particles and `j` also includes indices of *ghost* particles. Note that if `i` and `j` are both indices of *real* particles then the functor will be called with `f(i, j)` and `f(j, i)`.

```
template <typename ExecutionSpace, typename Functor>
void forEachPair(Functor&& f) const;
```

As mentioned above, iterating over all particles and then iterating over all of its neighbors is not the most efficient way. Instead we iterate over all non-*ghost* cells, then iterate over all neighboring cells. For all cell pairs we iterate over all combinations of particles between these two cells. Note that this might result in particle pairs which have a distance larger than r_{cutoff}. If this these pairs should be excluded, the functor `f` needs to take care of it.

To iterate over all cell neighbors (including the cell itself) we need to consider 3^D cells. As each cell is a neighbor, their D dimensional index differs by -1, 0 or 1 in every dimension. We can iterate over all the combinations by counting up to 3^D and treating the index as a ternary number with D digits. Each digit of a ternary number is 0, 1 or 2, thus shifting it by -1 gives us exactly the desired offsets. Listing 3.7 summarizes this procedure.

Listing 3.7: Computation of cell neighbor indices. The offsets are computed using ternary representation of the index while working with the D dimensional cell index.

```
constexpr size_type numNeighbors = detail::countHypercubes(Dim); // 3^D
for (size_type i = 0; i < numNeighbors; ++i) {
    auto temp = i;

    // extract the offsets from the base-3 representation of the index
    auto neighborCellIndex = cellIndex;
    for (unsigned d = 0; d < Dim; ++d) {
        neighborCellIndex(d) += (temp % 3) - 1;
        temp /= 3;
    }

    neighborIndices[i] = toFlatCellIndex(neighborCellIndex, ...);
}
```

With the cell neighbor computation done we can now look at the full particle iteration as shown in listing 3.8. Note that the outermost Kokkos for loop iterates only over non-*ghost* cells. But when considering the their neighbors ghost cells are automatically considered but only as second indices for the functor `f`.

To exploit the maximum level of parallelism we use Kokkos' nested parallelism in listing 3.8. There are three levels in Kokkos' parallelism hierarchy we are using and want to explain:

- `Kokkos::TeamPolicy` assigns team of threads to each loop iteration.
- `Kokkos::TeamThreadRange` this level of parallel loops splits an index range over the threads of a team.
- `Kokkos::ThreadVector(MD)Range` represents tightly nested parallelism and splits the work among the vector lanes of a thread. The range can be one or multidimensional (`MD`). On GPUs, vector lanes may correspond to the number of data elements that can be processed simultaneously within a warp. In contrast, the number of vector lanes on CPUs may correspond to the width of SIMD registers.

Listing 3.8: Iteration over all particle pairs. Nested parallelism is employed for maximum efficiency. The variable `numLocalCells_m` is the amount of non-*ghost* cells. The function `getCellNeighbors` performs the computations shown in listing 3.7.

```
constexpr auto numCellNeighbors = detail::countHypercubes(Dim); // 3^D

using team_policy_t       = Kokkos::TeamPolicy<ExecutionSpace>;
using team_t              = team_policy_t::member_type;
using team_thread_range_t = Kokkos::TeamThreadRange;
using vector_range_2d_t   =
↪  Kokkos::ThreadVectorMDRange<Kokkos::Rank<2>, team_t>

Kokkos::parallel_for(team_policy_t(numLocalCells_m, Kokkos::AUTO()),
↪  KOKKOS_LAMBDA(const team_t& team) {
    const size_type cellIdxFlat = team.league_rank();

    const auto cellParticleOffset = cellStartingIdx(cellIdxFlat);
    const auto numCellParticles   = cellParticleCount(cellIdxFlat);
    if (numCellParticles == 0) {
        return;
    }

    // Get all cell neighbors
    const auto cellNeighbors = getCellNeighbors(cellIdxFlat, ...);

    // Iterate over all cell neighbors
    Kokkos::parallel_for(team_thread_range_t(team, numCellNeighbors),
    ↪  [&](const size_t& n) {
        const auto nCellIdx            = cellNeighbors[n];
        const auto nCellParticleOffset = cellStartingIdx(nCellIdx);
        const auto numNCellParticles   = cellParticleCount(nCellIdx);

        // Iterate over all combinations of this cell's and
        // the neighboring cell's particles
        Kokkos::parallel_for(vector_range_2d_t(team, numCellParticles,
        ↪  numNCellParticles), [&](const size_t& i, const size_t& j) {
             const auto particleIdx = cellParticleOffset + i;
             const auto neighborIdx = nCellParticleOffset + j;

             if (neighborIdx == particleIdx) {
                 return;
             }

             // Call the functor
             f(particleIdx, neighborIdx);
         });
    });
});
```

3.1.5 Sorting Particles

To sort particles we need to sort all their attributes. Instead of sorting all attributes at the same place where we have computed the new indices for the particles (see listing 3.4) we can apply this permutation of indices to every attribute. IPPL provides the utility function `forAllAttributes` in the `ParticleBase` class to call a functor for every argument. The type of the attributes obtained in the functor are `ParticleAttribBase&`. Thus we added a new function to the abstract base class `ParticleAttribBase` and to `ParticleAttrib` to sort an attribute according to an index permutation: `applyPermutation(const hash_type&)`. It works by creating a copy and filling the values in according to the new index. Then the temporary is copied back to the attribute view. Listing 3.9 shows the algorithm in detail. One could think about instead of copying back the temporary just swapping the views. However one should be careful with the over allocation of `ParticleAttribute` and its implementation.

Listing 3.9: Implementation of `applyPermutation(...)`. Note that this is not very cache friendly.

```
void ParticleAttrib::applyPermutation(const hash_type& permutation) {
    const auto view = this->getView();
    const auto size = this->getParticleCount();

    view_type temp("copy", size);

    using policy_type = Kokkos::RangePolicy<execution_space>;
    Kokkos::parallel_for(policy_type(0, size), KOKKOS_LAMBDA(const
    ↪  size_type& i) {
        temp(permutation(i)) = view(i);
    });
    Kokkos::fence();

    Kokkos::deep_copy(Kokkos::subview(view, Kokkos::make_pair(0u,
    ↪  size)), temp);
}
```

The new function can then be used as in listing 3.10.

Listing 3.10: Sort all attributes according to `newIndex`. Makes sure to include attributes from all memory spaces.

```
detail::runForAllSpaces([&]<typename MemorySpace>() {
    const auto newIndexMirror = Kokkos::create_mirror_view_and_copy(
             MemorySpace(), newIndex);
    pc.template forAllAttributes<MemorySpace>(
        [&]<typename Attribute>(Attribute& att) {
            att->applyPermutation(newIndexMirror);
        });
});
```

3.1.6 Periodic Boundary Conditions

Periodic boundary conditions potentially introduce interactions over the global domain boundary. Obviously these interactions are not accounted so far as the particles are far apart. To also account for them, we can crate nearest image copies of particles which are close to the global domain boundary. While these nearest image copies will lay outside of the global the domain, they are within the overlap of some processes. Thus when creating these new particles they will only ever be *ghost* particles and invalidated immediately by the creating process if they are not within its overlap. Note that these nearest image copies need to be created before the particle exchange step from section 3.1.2.

First the particle indices which are close to boundaries with periodic boundary conditions are determined. A particle is considered close to the boundary, if it is within a distance of r_c to the boundary. Then we need to copy these particles.

To copy particles we need to copy all of its attributes. Due to the same reasons for the function `applyPermutation(...)` from section 3.1.5 we create a new function `internalCopy(const hash_type &hash)` which copies all attributes with indices from the hash. The copies are appended at the end of the view memory. The function can then be called for all attributes similar to listing 3.10. Now we can increase the number of particles stored by the particle container by the amount of copied particles. Otherwise they would not be considered during the particle exchange step.

Listing 3.11: Implementation of `internalCopy(...)`. Note that new particles are created but the particle count of the particle container will not be increased.

```
void ParticleAttrib::internalCopy(const hash_type &indices) {
    auto copySize = indices.size();
    create(copySize); // allocate more space if needed

    auto view = this->getView();
    const auto size = this->getParticleCount();

    using policy_type = Kokkos::RangePolicy<execution_space>;
    Kokkos::parallel_for(policy_type(0, copySize), KOKKOS_LAMBDA(const
    ↪  size_type &i) {
        view(size + i) = view(i);
    });
}
```

Finally we need to update their positions to their nearest periodic image. For every dimension with periodic boundary conditions we can correct all position components of the copied particles. The computation of the nearest image positions is shown in listing 3.12.

Listing 3.12: Position correction of the copied particles to their nearest periodic image. The variable `numLocParticles` is the number of particles stored on this process before coping boundary particles, `numBoundaryParticles` is the number of copied particles.

```
for (unsigned d = 0; d < Dim; ++d) {
    if (!periodic[d]) {
        continue;
    }
    const auto length = globalRegion[d].length();
    const auto middle = globalRegion[d].min() + length / 2;

    using range_t = Kokkos::RangePolicy<position_execution_space>;
    Kokkos::parallel_for(range_t(numLocParticles, numLocParticles +
    ↪  numBoundaryParticles), KOKKOS_LAMBDA(const size_t& i) {
        positions(i)[d] += positions(i)[d] > middle ? -length : length;
    });
}
```

3.2 Particle-Particle Interaction

To handle the particle-particle interaction a new type of solvers are introduced, interaction solvers. Their architecture is based on the Poisson solvers. We provide an absteact base class `ParticleInteractionBase` which manages all parameters of the solver and has the pure virtual function `void solve() = 0;`. All future particle particle interaction classes are intended to inherit from `ParticleInteractionBase` and implement the function `solve()`. While we have talked about force splitting in eq. (2.5) the Poisson solvers solver for fields (e.g. the electric field). With the relation

$$\mathbf{E}_i = \frac{\mathbf{F}_i}{q_i},$$

with $\mathbf{E}_i$ the electric field felt by particle i, $\mathbf{F}_i$ the force and q_i the particles charge, the electric field can directly be computed from the force. Thus the output of the particle interaction solvers is also the field felt on the particles.

We implement interaction due to the short range force from section 2.1.2 within the class `TruncatedGreenParticleInteraction` to match the name of the corresponding Poisson solver for the long-range part. The class is rather simple, it has a constructor which taking all relevant objects to compute the short-range interaction as given in listing 3.13 and implements the `solve` functions.

Listing 3.13: Constructor of `TruncatedGreenParticleInteraction`. The attribute `F` is the field felt by each particle. This is where the resulting contributions are written to during `solve`. The attribute `R` are the positions and the attribute `QM` can be charges, masses or something equivalten. In the parameter list there should be the parameters `"alpha"` (the parameter α from eq. (2.6)), `"rcut"` (the cutoff radius r_c) and `"force_constant"` (the force constant k_e for Coulomb forces of G for gravitational forces).

```
TruncatedGreenParticleInteraction(const ParticleContainer& pc,
↪  VectorAttribute& F, const VectorAttribute& R, const
↪  ScalarAttribute& QM, const ParameterList& params)
```

`TruncatedGreenParticleInteraction` assumes that the particle container contains a particle layout which provides the function `forAllPairs`. Then the computation of the particle particle interaction is quite simple as shown in listing 3.14. Note that the call to `solve()` assumes that the field is initialized somehow. In the P^3M method the long-range interaction can be calculated before the short-range interaction and thus the field is just the long-range part. If no long-range part is whished, then remember to set the field to zero before calling `solve()`.

Listing 3.14: The `solve()` function of `TruncatedGreenParticleInteraction`. Note that the `solve()` assumes that the field has been initialized somehow (usually by some long-range interaction).

```
void TruncatedGreenParticleInteraction::solve() {
    const auto& particleLayout = this->pc_m.getLayout();

    particleLayout.template
    ↪ forEachPair<execution_space>(KOKKOS_LAMBDA(const size_t& i,
    ↪ const size_t& j) {
        const Vector_t dist_ij = R(i) - R(j);
        const Scalar_t rsq_ij  = dist_ij.dot(dist_ij);
        if (rsq_ij >= rcut2) {
            return;
        }

        const auto Field_ij = // Field computation
        Kokkos::atomic_sub(&Field(i), Field_ij);
    });
}
```

3.3 P^3M Method

A single step of the P^3M method is summarized in listing 3.15.

Listing 3.15: The `solve()` function of `TruncatedGreenParticleInteraction`. Note that the `solve()` assumes that the field has been initialized somehow (usually by some long-range interaction).

```
pc->update();      // Exchange particles and apply Boundary conditions

par2grid();        // Map particles to grid
fsolver->solve(); // Poisson Field solver

grid2par();        // Map grid onto particles
isolver->solve(); // Particle Interaction solver

// Advect Particles according to some time stepping scheme
```

Chapter 4

Disorder Induced Heating

To verify the correctness of all the modifications, we aim to reproduce the results from Schwab [9]. Schwab and previously also Ulmer [8] tested their algorithms on the disorder induced heating (DIH) problem [2].

4.1 Background

Coulomb collisions significantly affect beam dynamics simulations. After electron gun injection, low-energy beams experience shot noise effects [9]. This study examines how particle collisions influence beam heating, relevant both post-injection and in storage rings.

Known from ultracold plasma physics, DIH occurs when ions start at random positions with negligible potential energy [2]. Due to energy conservation:

$$E_{\text{total}} = E_{\text{kinetic}} + E_{\text{potential}} \equiv \text{constant}$$

As correlation energy builds up, ions heat up. Less structured initial states produce more heating during the transition from disorder to Coulomb-structured states. Thus this phenomena is called "Disorder-Induces Heating" [24, 25]. For more background and details read [9, chapter 5].

4.2 Experimental Setup

Following previous work [8, 9], we simulate a realistic beam near the electron gun: a spherical coasting beam (radius $R = 17.74\,\mu\text{m}$) carrying 25 fC of electron charge (equivalent to $N = 156{,}055$ electrons) over 5 plasma periods. Constant focusing is applied via a linear force field toward the beam center, with magnitude of the average space charge force. Particles initialize at rest with uniformly distributed positions within the sphere.

4.2.1 Plasma Parameters

The plasma frequency is calculated as:

$$\omega_p = \sqrt{\frac{Ne^2}{m_0\varepsilon_0}} \approx 1.45 \times 10^{11}\,\text{s}^{-1}, \quad (145\,\text{GHz})$$

with N the number of particles, e the electron charge, m_0 the electron mass and ϵ_0 the electric permittivity. This corresponds to a plasma period of:

$$\tau = \frac{2\pi}{\omega_p} \approx 43\,\text{ps}.$$

The simulation runs for 5 periods which corresponds to a total simulation time of 215 ps. The time step of $\Delta t = 215\,\text{fs}$ is chosen to be sufficiently small due to potentially large relative momenta from collisions.

4.2.2 Computational Domain

The long-range solver currently only implements periodic boundary conditions. To approximate open boundary conditions as in [2, 8], the beam is placed in a large cubic domain with edge length $L = 100\,\mu\text{m}$. The computational mesh for the PM step has width $h = 0.39\,\mu\text{m}$, corresponding to a 256^3 grid. The particle-particle cutoff radius is varied from $1h$ to $8h$ to study accuracy effects, with splitting parameter $\alpha = \frac{2}{r_c}$.

4.3 Results

4.3.1 Verifying Correctness

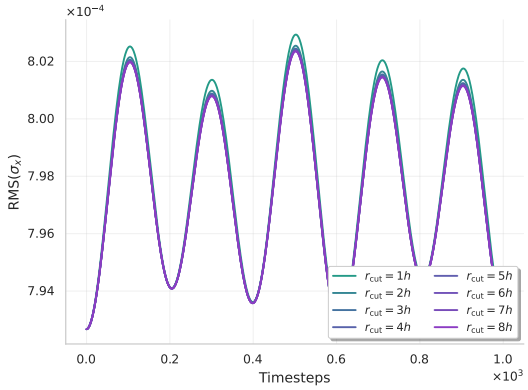
For the analysis of the disorder induced heating problem, each step we write out

- RMS beam size σ in x, y and z direction,
- RMS beam emittance ϵ in x, y and z direction,
- Temperature in x, y and z direction, and
- Kinetic energy of the particles.

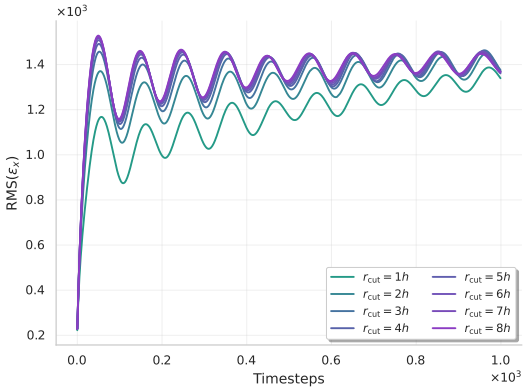
The main plot to reproduce is the relation between beam emittance and the cutoff radius r_c, see Schwab [9, chapter 5]. To verify the generalizations still provide the same results we run simulation on CPUs and Nvidia GPUs and with different amount of MPI ranks (nodes). All simulations are run on the Merlin 6 cluster. Figure 4.1 shows the results of the simulation run with 8 MPI ranks with one A100 GPU on each rank. Figure 4.2 shows the results of the simulation run across 4 MPI ranks with 8 CPUs on each rank. To compare our results with Schwab's results we copied a representative plot from his thesis, see fig. 4.3. Both results agree well with fig. 4.3 and confirm Schawb's findings; simulations with larger cutoff radii typically yield greater accuracy because they account for more direct particle interactions. As the plasma approaches its final period, fluctuations converge within a narrower range, and it becomes evident that accuracy gains plateau beyond a $3h$ cutoff radius. This behavior has a clear physical basis: the disorder-induced heating scenario involves particle collisions that cannot be adequately captured by Particle-in-Cell (PIC) methods. These "collisions" refer to Coulomb scattering events where particles never physically contact but deflect due to the rapidly intensifying interaction potential at close range. Once all relevant collision events are captured within a given cutoff radius, further expansion reveals no additional dynamics, explaining why accuracy improvements diminish beyond some threshold.

If we look closely at figs. 4.1 and 4.2 in contrast to fig. 4.3 we can see that our results show a more linear trend in the temperature and beam emittance. Meaning the oscillations fluctuate evenly around a linear trend line. While in Schwab's results fig. 4.3 the oscillations are not as uniform. Ulmer's results [8, Figure 8.1] confirm that the oscillations should be uniformly around

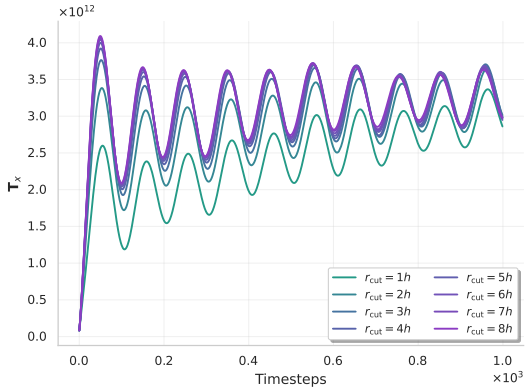
a linear trend. This finding suggests Schwab was missing some interactions and we have managed to include them again.



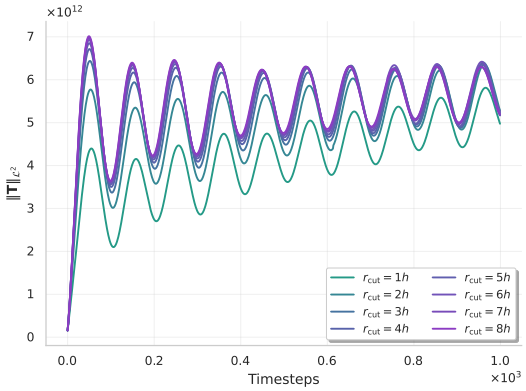
(a) RMS Beam Size in x vs Timesteps



(b) RMS Emittance in x vs Timesteps

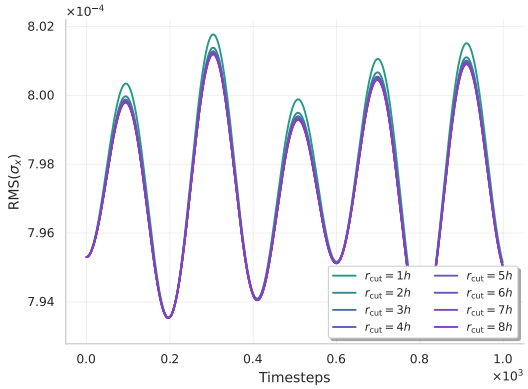


(c) Temperature in x vs Timesteps

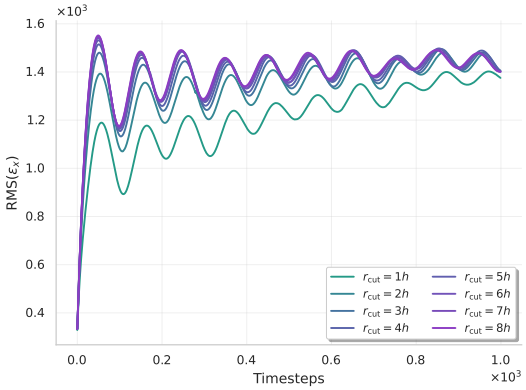


(d) L2-Norm of Temperature vs Timesteps

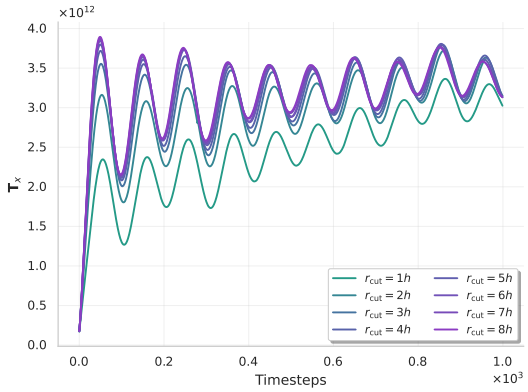
Figure 4.1: Disorder induced heating simulation on **GPU** over 5 plasma periods showing the evolution of beam parameters for different cutoff radii r_c. The simulation is run with 8 MPI ranks with one A100 GPU per rank. The simulations demonstrate the impact of varying cutoff parameters on (a) RMS beam size, (b) RMS emittance, (c) temperature, and (d) L^2-norm of temperature over time.



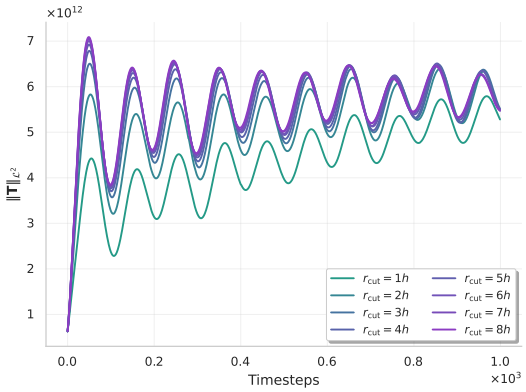
(a) RMS Beam Size in x vs Timesteps



(b) RMS Emittance in x vs Timesteps



(c) Temperature in x vs Timesteps



(d) L2-Norm of Temperature vs Timesteps

Figure 4.2: Disorder induced heating simulation on **CPU** over 5 plasma periods showing the evolution of beam parameters for different cutoff radii r_c. The simulation is run with 4 MPI ranks with 8 CPUs per rank. The simulations demonstrate the impact of varying cutoff parameters on (a) RMS beam size, (b) RMS emittance, (c) temperature, and (d) L^2-norm of temperature over time.

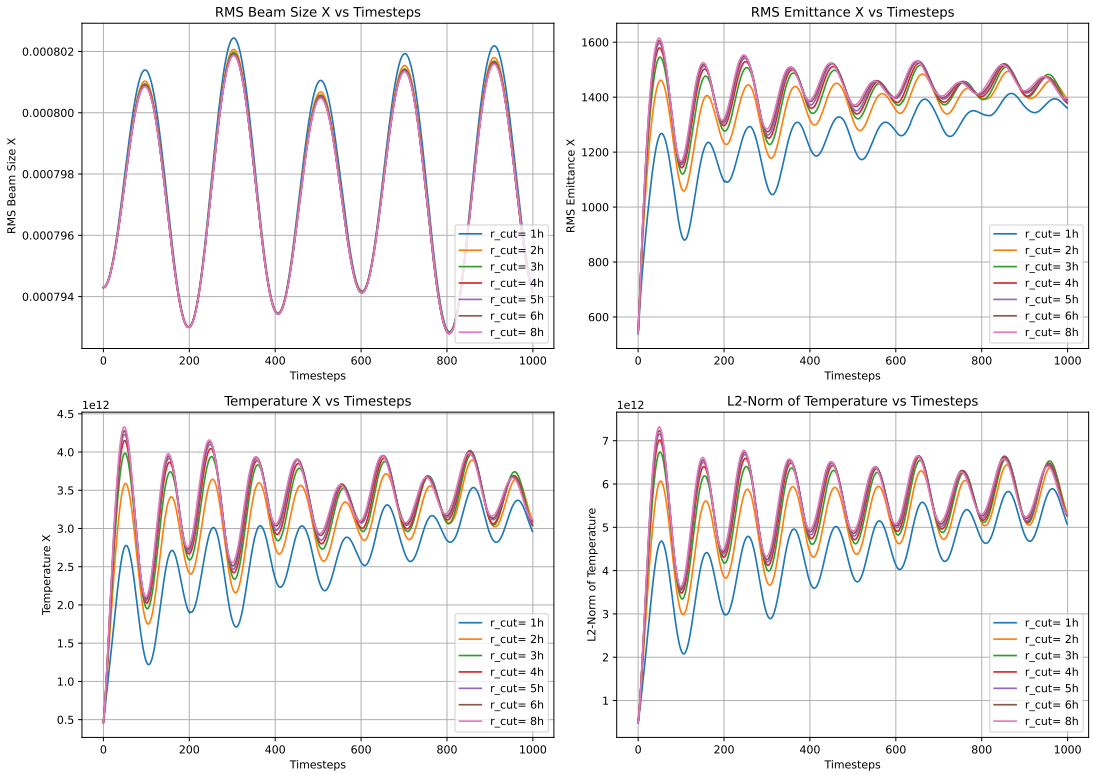


Figure 4.3: Disorder induced heating simulation of Schwab over 5 plasma periods showing the evolution of beam parameters for different cutoff radii r_c. The simulation is run with 8 MPI ranks with one A100 GPU per rank. Plotted are in (top left) RMS beam size, (top right) RMS emittance, (bottom left) temperature, and (bottom right) L^2-norm of temperature over time. The plot is directly copied from the thesis of Schwab [9, Figure 5.7].

4.3.2 Benchmarking

To test performance (portability), we perform strong and weak scaling benchmarks for the DIH problem on CPUs and GPUs.

Strong Scaling First for the strong scaling setup, the same parameters are used as when verifying the results, section 4.3.1, but with a cutoff radius $r_c = 1h$. Thus the simulations contain $156'055$ particles and a mesh grid of size 256^3 for the FFT. We can observe quit good strong scaling in the CPU case, figs. 4.4(a) and 4.4(b). The particle-particle interaction timers scale almost perfectly linearly with more tasks. Only the `updateParticle` timer shows no scaling at all. This is not surprising though as `updateParticle` times only the exchange of particles between different ranks. Thus with more ranks we can even expect this step to take longer. This is also the reason why the graph of `updateParticle` starts only two tasks, because for one task this timer is exactly zero and the equation for strong scaling

$$S(n) = \frac{T_0}{T(n)}$$

breaks down.

On GPU the strong scaling does not look as great. While `PPInteraction` timer scales perfectly `cellBuildTimer` does not. As `PPInteraction` contains only a easily parallelizeable double for-loop over all relevant particle pairs with some atomics we can expect this kind of scaling. Note that the atomics have at most 2 threads which write to the same location and hence we don't have too much congestion. On the other hand, the `cellBuildTimer` on the other hand contains some communication as well. With this setup with *only* 156'055 particles the overhead for the communication dominates the runtime for this step. We can observe this in table 4.1 i.e. the `cellBuildTimer` taking anly a fraction of the time for the complete particle-particle interaction for the single GPU run. Same reason for `updateParticle` timer's performance as for CPUs.

Table 4.1: Absolute wall-avg times (seconds) for 1 task and max tasks for the **strong scaling** runs. AMD EPYC 7713 CPUs and NVIDIA A100 GPUs were used.

Run	Tasks	`Total`	`PPInteraction`	`cellBuildTimer`	`updateParticle`
Single Node CPU	1	5'260	32.4	30.8	0
Single Node CPU	9	507	2.23	3.53	2.88
Multi Node CPU	9	514	2.49	2.79	3.6
GPU	1	40.4	18.3	2.04	0
GPU	4	90.1	4.62	2.13	3.51

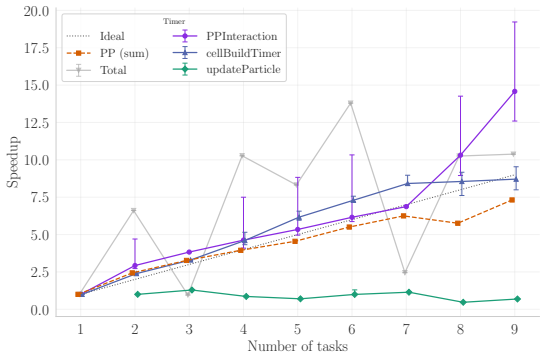
Weak Scaling For the weak scaling benchmark we again used the same parameters as when verifying the results, section 4.3.1, but with a cutoff radius $r_c = 1h$, however we changed the number of particles in the simulation depending on the number of tasks. For the CPU runs we used 150'000 particles per task and for the GPU runs we used 500'000 particles per task. However note that we still used a mesh of size 256^3 for the FFT. As we take the cutoff radius to be equal to the mesh width the term weak scaling should be taken with a bit of care because the number of cells for the neighbor list does not increase with more tasks. And the long range interaction (FFT) is not scaling weakly either. With this in mind we can understand why the weak scaling efficiency of `cellBuildTimer` for the CPU runs increases above one with more tasks. The other timers for the CPU runs stay around an efficiency of one. Besides again `updateParticles` which is a bit lower. Again this is to be expected as `updateParticles` exactly measures the overhead due to communication. Interestingly, looking at table 4.2 we can see that `updateParticles` takes longer on a single node vs communicating the particles via multiple nodes.

For the GPU runs we can observe decent efficiency up to four tasks. However at 5 GPUs `updateParticles` takes significantly longer (see table 4.2), which decreases the efficiency a lot. This can be explained by the fact that using more than four GPUs requires at least two nodes on the merlin cluster. On the other hand we have efficiency above one for the `PPInteraction` step. This might be a result from the number of cells not increasing as then each GPU needs to search through less cells to find all neighbors.

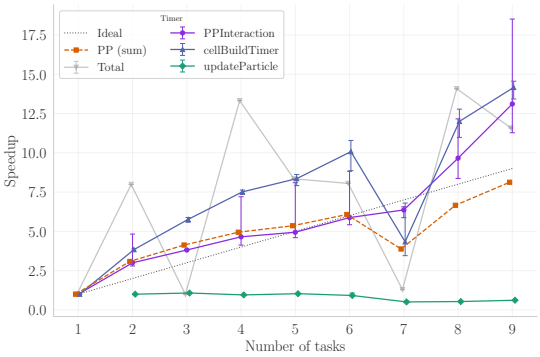
Conclusion We need benchmarks involving more tasks and better weak scaling i.e. the number of neighbor cells should increase as well. Another big thing to keep in mind is that in this DIH problem setup the particles are in the center of the domain while the work decomposition works spatially thus there might be ranks who have significantly less particles. A test setup with uniform distribution of particle might work better for benchmarking.

Table 4.2: Absolute wall-avg times (seconds) for 1 task and max tasks for the **weak scaling** runs. AMD EPYC 7713 CPUs and NVIDIA A100 GPUs were used.

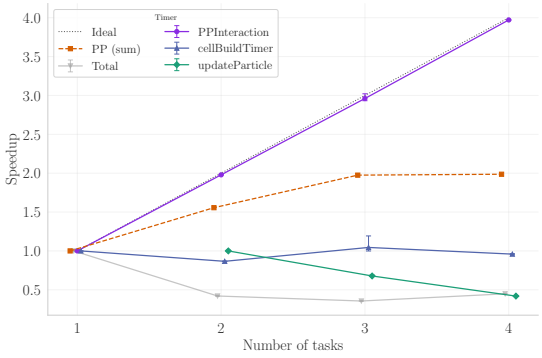
Run	Tasks	Total	PPInteraction	cellBuildTimer	updateParticle
Single Node CPU	1	5.260	35.9	27	0
Single Node CPU	10	682	39.2	4.38	21
Multi Node CPU	10	460	11.7	2.55	14.3
GPU	1	45.6	22.8	2.58	0
GPU	5	604	21.2	2.63	95.9



(a) Strong scaling on AMD EPYC 7713 CPUs. Each task (MPI rank) uses 8 CPUs and all tasks are on one node.

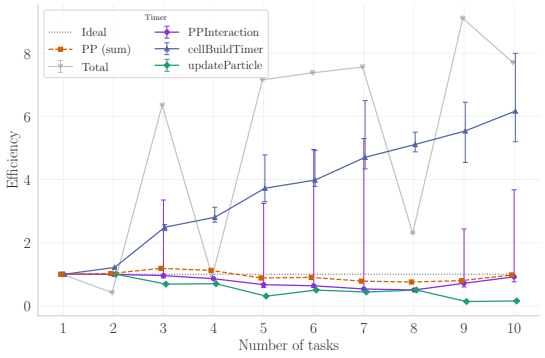


(b) Strong scaling on AMD EPYC 7713 CPUs. Each task (MPI rank) uses 8 CPUs each task is on a different one node.

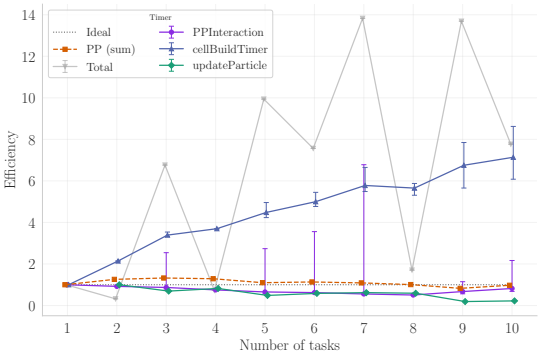


(c) Strong scaling on NVIDIA A100 GPUs. Each task (MPI rank) uses one GPU. All GPUs are on the same node.

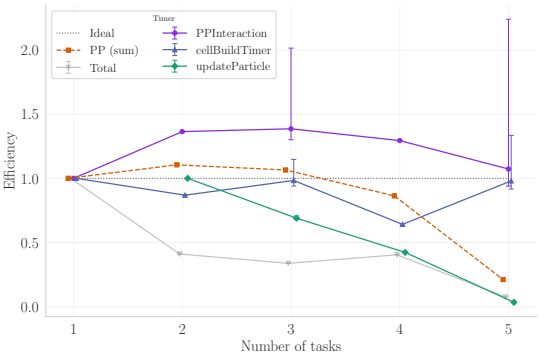
Figure 4.4: Strong scaling benchmarks using both CPUs (figs. 4.4(a) and 4.4(b)) and GPUs (fig. 4.4(c)). The setup involves 156'055 particles and a 256^3 grid for the FFT and using a cutoff radius $r_c = 1h$. Different timers related to the particle-particle interaction are plotted including the total simulation time. "PP (sum)" denotes the some of all particle-particle interaction relevant timers. The error bars are from the max and min times of IPPLs timers.



(a) Weak scaling on AMD EPYC 7713 CPUs. Each task (MPI rank) uses 8 CPUs and all tasks are on one node. There are 150'000 particles in the simulation per task.



(b) Weak scaling on AMD EPYC 7713 CPUs. Each task (MPI rank) uses 8 CPUs each task is on a different one node. There are 150'000 particles in the simulation per task.



(c) Strong scaling on NVIDIA A100 GPUs. Each task (MPI rank) uses one GPU. All GPUs are on the same node. There are 500'000 particles in the simulation per task.

Figure 4.5: Weak scaling benchmarks using both CPUs (figs. 4.5(a) and 4.5(b)) and GPUs (fig. 4.5(c)). The setup involves a 256^3 grid for the FFT and using a cutoff radius $r_c = 1h$. Different timers related to the particle-particle interaction are plotted including the total simulation time. "PP (sum)" denotes the some of all particle-particle interaction relevant timers. The error bars are from the max and min times of IPPLs timers.

4.4 Reproducing

To run the simulations on GPU on the Merlin 6 cluster, the run script from listing 4.1 can be used. If simulations on CPU want to be run, drop the `--cluster` and `--partition` options.

Listing 4.1: Run script to submit the simulations with `sbatch` on Merlin 6. This is for GPU submissions. The `cutoff_factor` can be changed to set the cutoff radius in relation to the PM mesh width h.

```
#!/bin/bash

#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=01:00:00
#SBATCH --cluster=gmerlin6
#SBATCH --partition=gwendolen

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export OMP_PROC_BIND=spread
export OMP_PLACES=threads

cutoff_factor=1

cd examples/collisions
srun ./P3MHeating 256 256 256 ${cutoff_factor} --info 10
↪  -device-id-by=mpi_rank
```

Chapter 5

Conclusion

We have incorporated the P^3M method into IPPL. While doing so we have managed to reproduce Schwab's results and improved them to match Ulmer's results more accurately. The treatment of particle-particle interactions in P^3M allows to more accurately describe a system of charged particles (i.e. DIH) than PIC methods while maintaining the computationally efficiency of PIC given small enough cutoff radii.

Some future steps are summarized below:

- Integrate the P^3M method within OPALX. To this end particle containers need the new `ParticleSpatialOverlapLayout` as their layout and the PP step needs to be applied after the field solvers and before advecting the particles.
- Since the P^3M method can now be run with any number of processes, perform scaling studies to identify potential bottlenecks.
- As the overlap introduces more communication between processes (more particle sent between processes), hiding this communication behind computation could be a feasible strategy. However this might require new strategies to perform the ghost particle exchange and the construction of the particle neighbor list.
- Implement open boundary conditions for the long-range part. similar to the FFT open solver.
- Perform benchmarks over a greater amount of GPUs and figure out the proper way to perform weak scaling i.e. make sure the number of neighbor list cells increases as well.

Bibliography

[1] R. Hockney and J. Eastwood, *Computer Simulation Using Particles* (CRC Press, Mar. 2021).

[2] C. Mitchell and J. Qiang, "A Parallel Particle-Particle, Particle-Mesh Solver for Studying Coulomb Collisions in the Code IMPACT-T", Proceedings of the 6th Int. Particle Accelerator Conf. **IPAC2015**, edited by H. Stuart (Ed.), A. Evelyn (Ed.), S. Todd (Ed.), and S. R.W. (Ed.) Volker, 3 pages, 0.636 MB (2015).

[3] T. Theuns, A. Leonard, G. Efstathiou, F. R. Pearce, and P. A. Thomas, "P3M-SPH simulations of the Lya forest", Monthly Notices of the Royal Astronomical Society **301**, 478–502 (1998).

[4] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald: An N log(N) method for Ewald sums in large systems", The Journal of Chemical Physics **98**, 10089–10092 (1993).

[5] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. In 'T Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales", Computer Physics Communications **271**, 108171 (2022).

[6] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers", SoftwareX **1-2**, 19–25 (2015).

[7] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", Mathematics of Computation **19**, 297 (1965).

[8] B. Ulmer, "The P3M Model on Emerging Computer Architectures with Application to Microbunching", MA thesis (ETH, Zurich, 2016), https://amas.web.psi.ch/people/aadelmann/ETH-Accel-Lecture-1/projectscompleted/cse/thesisBUlmer.pdf.

[9] T. Schwab, "A Performance Portable Version of the P3M Algorithm", BSc Thesis (ETH, Zurich, 2024), https://amas.web.psi.ch/people/aadelmann/ETH-Accel-Lecture-1/projectscompleted/cse/Thesis_Timo_Schwab.pdf.

[10] A. Adelmann, P. Calvo, M. Frey, A. Gsell, U. Locans, C. Metzger-Kraus, N. Neveu, C. Rogers, S. Russell, S. Sheehy, J. Snuverink, and D. Winklehner, *OPAL a Versatile Tool for Charged Particle Accelerator Simulations*, 2019, https://arxiv.org/abs/1905.06654.

[11] A. Adelmann, *OPAL-X*, 2025, https://gitlab.psi.ch/OPAL/opal-x/src.

[12] M. Frey, A. Vinciguerra, S. Muralikrishnan, Sonali, vmontanaro, Mohsen, A. Adelmann, manuel5975p, and F. Schurk, *IPPL-framework/ippl: IPPL-3.2.0*, Mar. 2024, https://zenodo.org/doi/10.5281/zenodo.10878166.

[13]S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, and A. Adelmann, "Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures", in Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP) (2024), pp. 26–38.

[14]H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns", Journal of Parallel and Distributed Computing **74**, 3202–3216 (2014).

[15]C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The Kokkos Ecosystem: Comprehensive Performance Portability for High Performance Computing", Computing in Science Engineering **23**, 10–18 (2021).

[16]C. Birdsall and A. Langdon, *Plasma Physics via Computer Simulation* (CRC Press, Oct. 2018).

[17]L. Verlet, "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules", Physical Review **159**, 98–103 (1967).

[18]W. Mattson and B. M. Rice, "Near-neighbor calculations using a modified cell-linked list method", Computer Physics Communications **119**, 135–148 (1999).

[19]M. P. Howard, J. A. Anderson, A. Nikoubashman, S. C. Glotzer, and A. Z. Panagiotopoulos, "Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units", Computer Physics Communications **203**, 45–52 (2016).

[20]M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford University PressOxford, June 2017).

[21]P. H. Hünenberger, "Optimal charge-shaping functions for the particle–particle—particle–mesh (P3M) method for computing electrostatic interactions in molecular simulations", The Journal of Chemical Physics **113**, 10464–10476 (2000).

[22]V. Ballenegger, J. J. Cerdà, and C. Holm, "How to Convert SPME to P3M: Influence Functions and Error Estimates", Journal of Chemical Theory and Computation **8**, 936–947 (2012).

[23]A. Sutton and R. Smith, *Folding expressions*, Nov. 2014, `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4295.html`.

[24]D. Gericke and M. Murillo, "Disorder-induced heating of ultracold plasmas", Contributions to Plasma Physics **43**, 298–301 (2003).

[25]J. M. Maxson, I. V. Bazarov, W. Wan, H. A. Padmore, and C. E. Coleman-Smith, "Fundamental photoemission brightness limit from disorder induced heating", New Journal of Physics **15**, 103024 (2013).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Integration and Testing of the P3M Algorithm in IPPL

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Bachmann	Jonas

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
31.07.2025, Lauenburg, Germany	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.