Contents

Chapter 1

Method

1.1 Particle-in-Cell

To investigate the mechanics involved in a wide variety of plasma phenomenens, computer simulation. Particle-in-Cell, PiC, is a model that takes a particle based approach, where each particle is simulated seperately, or a collection. If we naively would attempt to compute the electrical force between each particle, the necessary computational power would grow quickly as the number of particle increases, $\mathcal{O}(\#particles^2)$. Since a large number of particles is often neccessary the PiC method seeks to the problem by computing the electrical field produces by the particles, and then compute the force on the particle directly from the field instead. From the charge distribution we can find the electric potential through the use of Poissons equation, eq. (1.1), and subsequently find the electrical field from the electrical potential. So an important part of a PiC simulation is an numerically efficient Poisson Solver.

$$\nabla^2\Phi = -\rho \qquad \text{in} \qquad \Omega \tag{1.1}$$

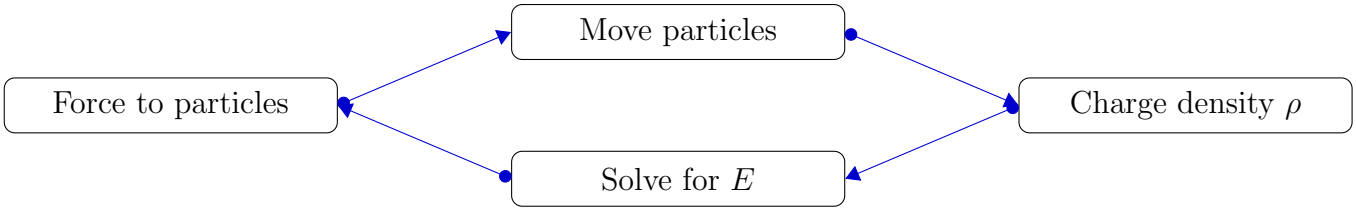


Figure 1.1: Schematic overview of the PIC method

1.1.1 Spectral Methods

The spectral methods is based on Fourier transforms of the problem and solving the problem in it's spectral version, see (Shen, 1994), for an implementation of an spectral poisson solver. They are efficient solvers that can be less intricate to implement (?), but can be inaccurate for complex geometries.

When looking for a solution with a spectral method we first rewrite the functions as Fourier series, which for the three-dimensional Poisson equation would be

$$\nabla^2 \sum A_{j,k,l} e^{i(jx+ky+lz)} = \sum B_{j,k,l} e^{i(jx+ky+lz)} \tag{1.2}$$

From there we get a relation between the coefficients

$$A_{j,k,l} = -\frac{B_{j,k,l}}{j^2 + k^2 + l^2} \tag{1.3}$$

Then we compute the Fourier transform of the right hand side obtaining the coefficients $B_{j,k,l}$. We compute all the coefficients $A_{j,k,l}$ from the relation between the coefficients. At last we perform a inverse Fourier transform of the left hand side obtaining the solution.

(1.4)

1.1.2 Finite Element Methods

The finite element is a method to numerically solve a partial differential equations (PDE) first transforming the problem into a variational problem and then constructing a mesh and local trial functions, see Alnæs et al., 2011 for a more complete discussion.

To transform the PDE to a variational problem we first multiply the PDE by a test function v, then it is integrated using integration by parts on the second order terms. Then the problem is separated into two parts, the bilinear form $a(u, v)$ containing the unknown solution and the test function and the linear form $L(v)$ containing only the test function.

$$a(u, v) = L(v) \qquad v \epsilon \hat{V} \tag{1.5}$$

Next we construct discrete local function spaces of that we assume contain the trialfunctions and testfunctions. The function spaces often consists of locally defined functions that are 0 except in a close neighbourhood of a mesh point, so the resulting matrix to be solved is sparse and can be computed quickly. The matrix system is then solved by a suiting linear algebra algorithm, before the solution is put together.

1.2 Multigrid

The multi grid, MG, method used to solve the Poisson equation and obtain the electric field is a widely used and highly efficient solver for elliptic equations, having a theoretical scaling of $\mathcal{O}(N)$ (Press et al., 1988), where N is the grid points. Here I will go through the main theory and algorithm behind the method, as explained in more detail in (Press et al., 1988; Trottenberg et al., 2000) as well as go through some of possible algorithms to parallelize the method.

We want to solve a linear elliptic problem,

$$\mathcal{L}u = f \tag{1.6}$$

where $\mathcal{L}$ is a linear operator, u is the solution and f is a source term. In our specific case the operator is given by the laplacian, the source term is given by the charge density and we solve for the electric potential.

We discretize the equation onto a grid of size q.

$$\mathcal{L}_q u_q = f_q \tag{1.7}$$

Let the error, v_q be the difference between the exact solution and an approximate solution to the difference equation (1.7), $v_q = u_q - \tilde{u}_q$. Then we define the residual as what is left after using the approximate solution in the equation.

$$d_q = \mathcal{L}_q \tilde{u}_q - f_q \tag{1.8}$$

Since $\mathcal{L}$ is a linear operator the error satisfies the following relation

$$\mathcal{L}_q v_q = \mathcal{L}(u_q - \tilde{u}_q) + (f_q - f_q) \tag{1.9}$$
$$\mathcal{L}_q v_q = -d_q \tag{1.10}$$

In the multigrid methods instead of solving the equation directly here, we set up a system of nested coarser square grids, $\mathfrak{T}_1 \subset \mathfrak{T}_2 \subset \cdots \subset \mathfrak{T}_\ell$, where 1 is the coarsest grid and ℓ is the finest. Then the main thought behind the methods is that instead of solving the problem directly on the initial grid, we use restriction, $\mathcal{R}$, and interpolation, $\mathcal{P}$, operators to change the problem between the different grid levels and solve them on there. (Fix previous sentence) Due to the fewer grid points the problem is faster to solver on the coarser grid levels than on the fine grid.

If we then apply a restriction operator on the residual we go down a level on the grids and the opposite for the interpolation operator.

$$\mathcal{R}d_q = d_{q-1} \qquad \text{and} \qquad \mathcal{I}d_q = d_{q+1} \tag{1.11}$$

1.2.1 Algorithm

A sequential algorithm for a two grid V shaped algorithm, where the coarse grid and fine grid has respectively $q = 1, 2$.

- Initial approximation. $\tilde{u}$
- for $i <$ nCycles:
  - Presmooth: $\hat{u}_2 = S_{pre}(u)_2$
  - Calculate defect: $d_2 = f_2 - \mathcal{L}\hat{u}_2$
  - Restrict defect: $d_1 = Rd_2$
  - Initial guess: $\tilde{u}_1 = 0$
  - Solve (GS-RB): $L_1\tilde{u}_1 = d_1$
  - Interpolate: $\tilde{u}_2 = I\tilde{u}_1$
  - Add correction: $u_2^{new} = \hat{u}_2 + \tilde{u}_2$

1.2.2 Smoothing: Gauss-Seidel

Relaxation methods, such as Gauss-Seidel, work by looking for the setting up the equation as a diffusion equation, and then solve for the equilibrium solution.

So suppose we want to solve the elliptic equation

$$\mathcal{L}u = \rho \tag{1.12}$$

Then we set it up as a diffusion equation

$$\frac{\partial u}{\partial t} = \mathcal{L}u - \rho \tag{1.13}$$

By starting with an initial guess for what u could be the equation will relax into the equilibrium solution $\mathcal{L}u = \rho$. By using a Forward-Time-Centered-Space scheme to discretize, along with the largest stable timestep $\Delta t = \Delta^2/(2 \cdot d))$, we arrive at Jacobi's method, which is an averaging of the neighbors in addition to a contribution from the source term. By using the already updated values for in the calculation of the u^{new} we arrive at the method called Gauss-Seidel which for two dimensions is the following

$$u_{i,j}^{n+1} = \frac{1}{4}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n} + u_{i,j-1}^{n+1}\right) - \frac{\Delta^2 \rho_{i,j}}{4} \tag{1.14}$$

A slight improvement of the Gauss-Seidel algorithm is achieved by updating every other grid point at a time, by using Red and Black Ordering. This allows a vectorization of the problem and avoids any uneccessary copying.

1.3 Parallelization

For the parallelization of an algorithm there is two obvious issues that need to be addressed to ensure that the algorithm retains a high degree of parallelization; communication overhead and load imbalance (Hackbusch and Trottenberg, 1982). Communication overhead means the time the computational nodes spend communicating with each other, if that is longer than the actual time spent computing the speed of the algorithm will suffer, and load imbalance appears if some nodes need to do more work than others causing some nodes to stand idle.

Here we will focus on multigrid of a 3D cubic grid, where each grid level has half the number of grid points. We will use grid partitioning to divide the domain, GS-RB (Gauss-Seidel Red-Black) as both a smoother and a coarse grid solver.

We need to investigate how the different steps: interpolation, restriction, smoothing and the coarse grid solver, in a MG algorithm will handle parallelization.

1.3.1 Grid Partition

There are several well explored options for how a multigrid method can be parallized, for example Domain Decomposition (**arraras˙domain˙2015**), Algebraic Multigrid (Stüben, 2001), see Chow et al. (2006) for a survey of different techniques. Here we will focus on Geometric Multigrid (GMG) with domain decomposition used for the parallelization, as described in the books Trottenberg et al., 2000; Hackbusch and Trottenberg, 1982.

With domain partitioning we divide the grid $\mathcal{T}$ into geometric subgrids, then we can let each processes handle one subgrid each. As we will see it can be useful when using the GS-RB smoothing, as well as other parts of a PiC program, to extend the subgrids with layers of ghost cells. The GS-RB algorithm will directly need the adjacent nodes, in its neighbour subdomain.

1.3.2 Distributed and accumulated data

(Remove? Not actually relevant for our implementation) During the parallel execution of the code there can be useful to keep track of the different data structures and what needs to be accumulated over all the computational nodes and what only needs to be distributed on the individual computational nodes.

- u solution (Φ)
- w temporary correction
- d defect

- f source term (ρ)
- $\mathcal{L}$ differential operator
- $\mathcal{I}$ interpolation operator
- $\mathcal{R}$ restriction operator
- $\mathbf{u}$ Bold means accumulated vector
- $\tilde{\mathbf{u}}$ is the temporary smoothed solution
- Accumulated vectors: $\mathbf{u}_q$, $\hat{\mathbf{u}}_q$, $\tilde{\mathbf{u}}_q$, $\hat{\mathbf{w}}_q$ $\mathbf{w}_{q-1}$, $\mathbf{I}$,$\mathbf{R}$
- Distributed vectors: f_q, d_q, d_{q-1}

1.3.3 Smoothing

We have earlier divided the grid into subgrids, with overlap, as described in subsection 1.3.1 and given each processor responsibility for a subgrid. Then do a a GS-RB method we start with an approximation of $u^n_{i,j}$. Then we will obtain the next iteration by the following formula

$$u^{n+1}_{i,j} = \frac{1}{4}\left(u^n_{i+1,j} + u^{n+1}_{i-1,j} + u^n_{i,j+1} + u^{n+1}_{i,j-1}\right) - \frac{\Delta^2 \rho_{i,j}}{4} \tag{1.15}$$

We can see that the for the inner subgrid we will have no problems since we have all the surrounding grid points. On the edges we will need the adjacent grid points that are kept in the other processors. To avoid the algorithm from asking neighboring subgrids for adjacent grid points each time it reaches a edge we instead update the entire neighboring column at the start. So we will have a 1-row overlap between the subgrids, that need to be updated for each iteration.

1.3.4 Restriction

For the transfer down a grid level, to the coarser grid we will use a half weighting stencil. In two dimensions it will be the following

$$\mathcal{R} = \frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{1.16}$$

With the overlap of the subgrids we will have the necessary information to perform the restriction without needing communication between the processors (Hackbusch and Trottenberg, 1982).

1.3.5 Interpolation

For the interpolation we will use bilinear interpolation:

$$\mathcal{I} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{1.17}$$

Since the interpolation is always done after GS-RB iterations the the outer part overlapped part of the grid updated, and we can have all the necessary information.

1.3.6 Scaling

Volume-Boundary effect

While a sequential MG algorithm has a theoretical scaling of $\mathcal{O}(N)$ (Press et al., 1988), where N is the number of grid points, an implementation will have a lower scaling efficiency due to interprocessor communication. We want a parallel algorithm that attains a high speedup with more added processors P, compared to sequential 1 processor algorithm. Let $T(P)$ be the computational time needed for solving the problem on P processors. Then we define the speedup $S(P)$ and the parallel efficiency $E(P)$ as

$$S(P) = \frac{T(1)}{T(P)} \qquad E(P) = \frac{S(P)}{P} \tag{1.18}$$

A perfect parallel algorithm would the computational time would scale inversely with the number of processors, $T(P) \propto 1/P$ leading to $E(P) = 1$. Due to the necessary interprocessor communication that is generally not achievable. The computational time of the algorithm is also important, if the algorithm is very slow but has good parallel efficiency it is often worse than a fast algorithm with a worse parallel efficiency.

The parallel efficiency of an algorithm is governed by the ratio between the time of communication and computation, T_{comm}/T_{comp}. If there is no need for communication, like on 1 processor, the algorithm is perfectly parallel efficient. In our case the whole grid is diveded into several subgrids, which is assigned to different processors. In many cases the time used for computation is roughly scaling with the interior grid points, while the communication time is scaling with the boundaries of the subgrids. If a local solution method is used on a local problem it is only the grid points at the boundary that needs the information from grid points on the other processors. Since the edges has lower dimensionality than the inner grid points. So when the problem is is increasing the time for

	Cycle	Sequential	Parallel
MG	V	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$
	W	$\mathcal{O}(N)$	$\mathcal{O}\left(\sqrt{N}\right)$
FMG	V	$\mathcal{O}(N)$	$\mathcal{O}\left(\log^2 N\right)$
	W	$\mathcal{O}(N)$	$\mathcal{O}\left(\sqrt{N}\log N\right)$

Table 1.1: The parallel complexities of sequential and parallel multigrid cycles

computation grows faster than the time for communication, and a parallel algorithm often has a higher parallel efficiency on a larger problem. This is called the Boundary-Volume effect.

Parallel complexity

The complexities, as in needed computational work, of sequential and parallel MG cycles is calculated in Hackbusch and Trottenberg, 1982 and is shown in table 1.1. In the table we can see that in the parallel case there is a substantial increase in the complexity in the case of W cycles compared to V cycles. In the sequential case the change in complexity when going to a W cycle is not dependent on the problem size, but it is in the parallel case.

1.4 Grid Structs and Partitioning

1.4.1 Data structures

The fields and quantities in our PiC model is discretized on a threedimensional grid and comes to use several places in the method. In the multigrid calculation we also have a use for several grids of varying spatial coarseness. So it will be useful for us to organize the data so we have the grid stored as an independent structure available for the program, while the multigrid part uses an extended version where it also has access to the different coarser subgrids. There will also be several types of grid, all with the same specifications, but storing different quantities, so we will use a grid template that all the grids of the same size shares, while we will have a seperate struct containing the quantities in the grid and the specifications.

1.4.2 Domain partitioning

As earlier discussed, in 1.3.1 the physical domain covered by our model will be divided onto several processors that each take care of a subdomain. The subdo-

mains are dependent on each other and we need some communication between them, which we solve by letting each subdomain also store the edge of the neighboring subdomain. Depending on the boundary conditions it could also be useful to store an extra set of values on the outer domain boundary as well, which will be called ghost points, N_G. The extra grid points due the the overlap between the subdomains we will call overlap points, N_O. Let us for simplicity sake consider a regular domain, with equal extent in all dimensions, with N grid points per dimension, d and consider how many grid values we need to store as a singular domain and the grid values needed when it is divided amongst several processors, a 2 dimensional case is depicted in fig. 1.2.

1.4.3 Singular domain

In the case where the whole domain is worked on by one process we need N^d to store the values on the grid representing the physical problem, in addition we see that we also need to store values for the ghost points along the domain boundary. Given that we have one layer of ghost points on all the boundaries, and there is 2 boundaries per dimension, the total number of ghost points is given by $N_G = 2dN$. Since there is only 1 domain we don't need to account for any overlap between subdomains and the total grid points we need to store is:

$$N_{Tot} = N^d + N_G + N_O = N^d + 2dN^{d-1} \tag{1.19}$$

For the 2 dimensional case, in fig. 1.2, that adds up to $N_{Tot} = 8^2 + 2 \times 4 \times 8 = 128$.

1.4.4 Several subdomains

In the case where we introduce several subdomain, in addition to storing the grid values and the ghost points we also need to store an overlap between the subdomains. If we take our whole domain Ω and divide in up into several small domains Ω_S, the smaller domains only takes a subsection of the grid points. For simplicity case, and equal load on processors, we let the subdomains as well be regular, with the whole domain being a multiple of the subdomains. Our whole domain has N grid points in each direction, if we then divide that domain into $\#\Omega$ domains, then each of those subdomains will have $N_S = N^d/\#\Omega$ grid points. Each of those subdomains will also need values representing the ghost points and overlap from the neighboring nodes. A boundary of a subdomain will either have overlap points, or ghost points, not both at the same time so for each boundary we need to 1 layer, N_S^{d-1}. Each subdomain will have 2 boundaries per dimension since we have regular subdomains. The total number of grid points needed per subdomain is then

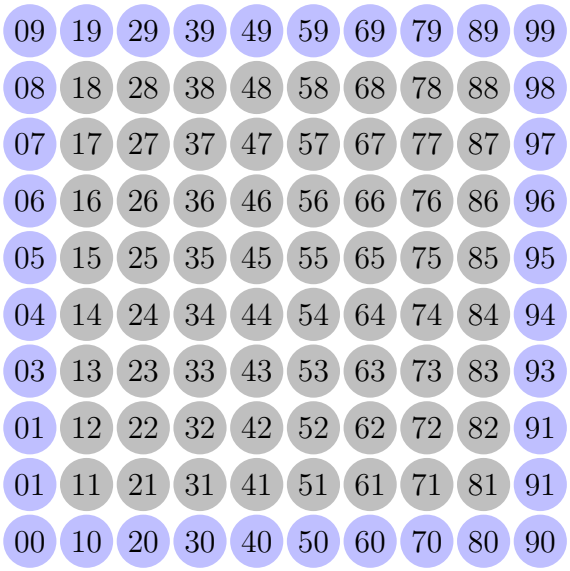
$$N_{Tot,S} = N_S^d + (N_G + N_O) = N_S^d + 2dN_S^{d-1} \tag{1.20}$$

while the total number of grid points is

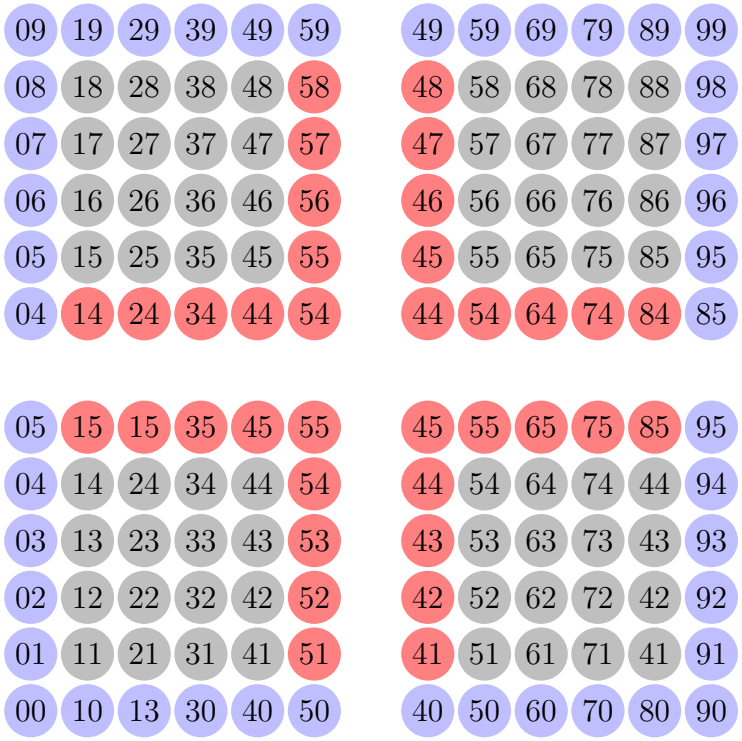
$$N_{Tot} = \#\Omega N_{Tot,S} \tag{1.21}$$

For the 2 dimensional case discussed earlier we need $N_{Tot,S} = 4^2 + 2 \times 2 \times 4^1 = 32$.

Since the effect of the subdomain boundaries increase the coarser the grid is we should not let the coarsest multigrid level be to small. We also don't need the spatial extent of the grid to be equal on all sides, but it was done here to keep the computations simple.



(a) The grid points needed for an 8×8 domain.



(b) The 8×8 grid divided into 4 subdomains

Figure 1.2: Each circle in the figures represents 1 grid point, and the first number is the column while the second is the row. The grey colour represents physical space the computational node works on, the blue color is the outer grid points for boundary conditions and the red colour is the overlapping grid points.