Chapter 1

Method

1.1 Poisson Solvers

In the Particle in Cell method, instead of calculating the forces directly between the particles, we first calculate the electric field from all the particles and then use that to calculate the individual force on the particles. From the particles we use a weighting scheme to obtain a charge density on a grid. Then we can use Poisson's equation eq. (1.1) to calculate the electric potential from the charge density, before computing the electric field from the potential. So we need an efficient Poisson solver.

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

The problem also need boundary conditions in which we will focus on periodic, Dirichlet and Neumann boundary conditions.

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˙efficient˙1994** for an implementation of an spectral poisson solver. They are efficient solvers that can be less, 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 **Sandve2011** 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 $\mathcal{O}(N)$ (**Press1987**), 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 (**Press1987**; **Trottenberg2000**) 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 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.

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

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: $d_1 = R d_2$
  - Initial guess: $\tilde{u}_1 = 0$
  - Solve (G-S 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 $\Delta t = \Delta^2/2^d$ (Double check the factor.), 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

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

To achieve a vectorization of the calculation of $u_{i,j}^{n+1}$ we will use Red and Black ordering, first calculating the odd nodes and then the even nodes, so the method has the available information.

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 (**Trottenberg**). 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 (to be expanded to rectangular cubes) 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 method 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 (**Arraras2015**), Algebraic Multigrid

(**StUben2001**), see **Chow2006** for a survey of different techniques. Here we will focus on Geometric Multigrid (GMG) with grid partitioning used for the parallelization, as described in the books **Trottenberg2000**; **Trottenberg**

With grid 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 G-S RB smoothing to let the subgrids overlap 1 layer deep. since it on the edges of the subgrid it will need the adjacent node values.

1.3.2 Distributed and accumulated data

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
- $\vec{\mathbf{u}}$ Bold means accumulated vector
- $\tilde{\vec{\mathbf{u}}}$ is the temporary smoothed solution
- Accumulated vectors: $\vec{\mathbf{u}}_q$, $\hat{\vec{\mathbf{u}}}_q$, $\tilde{\vec{\mathbf{u}}}_q$, $\hat{\vec{\mathbf{w}}}_q$ $\vec{\mathbf{w}}_{q-1}$, $\vec{\mathbf{I}}$,$\vec{\mathbf{R}}$
- Distributed vectors: f_q, d_q, d_{q-1}

Algorithm: P is the number of processes

- If (q == 1): Solve: $\sum_{s=1}^{P} \mathcal{L}_{s,1}\vec{\mathbf{u}}_1 = \sum_{s=1}^{P} f_{s,1}$
- else:
  - Presmooth: $\hat{\tilde{\mathbf{u}}}_q = \mathcal{S}_{pre}\vec{\mathbf{u_q}}$
  - Compute defect: $d_q = f_q - \mathcal{L}_q\hat{\tilde{\mathbf{u}}}_q$
  - Restrict defect: $d_{q-1} = \mathcal{R}d_q$
  - Initial guess: $\vec{\mathbf{w}}_{q-1} = 0$
  - Solve defect system: $\vec{\mathbf{w}}_{q-1} = PMG(\vec{\mathbf{w}}_{q-1}, d_{q-1})$
  - Interpolate correction: $\vec{\mathbf{w}}_q = \mathcal{R}\vec{\mathbf{w}}_{q-1}$
  - Add correction: $\tilde{\vec{\mathbf{u}}}_q = \hat{\vec{\mathbf{u}}}_q + \vec{\mathbf{w}}_q$
  - Post-smooth: $\vec{\mathbf{u}}_q = \mathcal{S}_{post}$

1.3.3 Smoothing

Will use G-S with R-B ordering, supposedly good parallel properties (**Chow2006**). Follow algorithm I in **Adams2001** (alternatively try to implement the more complicated version)?

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 $u_{i,j}^n$. Then we will obtain the next iteration by the following formula

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

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 (**Trottenberg**).

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 $\mathcal{O}(N)$ (**Press1987**), 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 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 **Trottenberg** 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.

	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