

Particle-in-cell plasma simulation with OmpSs-2

Rodrigo Arias Mallo

June 22, 2019

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Objectives	5
1.3	Context	5
1.4	Structure	5
2	Related work	7
I	Theory	9
3	Plasma introduction	11
4	Plasma simulation	13
4.1	The particle-in-cell method	13
4.2	Particle mover	13
4.2.1	Boris integrator	13
4.3	Charge accumulation	14
4.4	Field equations	14
5	Discrete model	15
5.1	Charge assignment	15
5.2	Field equations	16
5.2.1	Electric potential	16
5.2.2	LU decomposition	16
5.2.3	Multiple Fourier Transform (MFT)	17
5.2.4	Electric field	18
5.3	Force interpolation	18
5.4	Equations of motion	19
5.4.1	Boris integrator	19
II	Computation	21
6	Sequential simulator	23
6.1	Design	23
6.1.1	Debug mode	23

6.2	Validation	24
6.2.1	Two particle test	24
6.2.2	Two stream instability	25
6.2.3	Cyclotron frequency	25
7	Parallelization techniques	27
7.1	Message Passing Interface	27
7.1.1	Concepts	27
7.1.2	Implementations	28
7.2	OmpSs-2	28
7.2.1	Concepts	28
8	Simulator design	29
8.1	Field solver	29
9	The simulator	31
9.1	Data structures	31
9.2	Simulation flow	31
9.2.1	Initialization	32
9.2.2	Loop	33
9.2.3	Finish	33
10	Communication	35
10.1	Particle communication	35
10.1.1	Exchange in X	35
10.1.2	Exchange in Y	38
10.1.3	Mitigation of deadlocks with TAMPI	39
10.2	Field communication	42
10.2.1	Charge density ρ	42
10.2.2	Electric potential ϕ	43
10.2.3	Electric field \mathbf{E}	44
11	Analysis	45
11.1	Analysis with varying inputs	46
12	Analysis of performance	47
12.1	Variables of interest	47
12.1.1	Number of particles	47
12.1.2	Constant number of CPUs	47
13	Configuration	51
14	Caveats and limitations	53

Chapter 1

Introduction

It may be surprising to find out that the most common state of matter is plasma when we look at the universe. A plasma is an ionized gas consisting of ions and free electrons distributed over a region in space. in which at least one electron of the atom is separated, so it remains positively charged (ionized) [1]. Usually this happens in the vacuum

1.1 Motivation

1.2 Objectives

1.3 Context

1.4 Structure

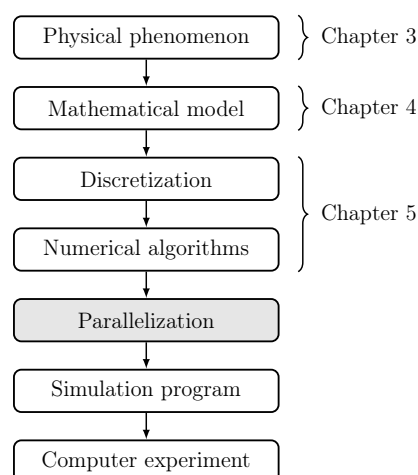


Figure 1.1: Principal steps in computer experiment

The structure of the document follows the diagram shown in the figure 1.1. In chapter 1, plasma is described as a physical phenomenon and we focus on the

relevant properties that we want to study, from which we derive a mathematical model. In chapter 4. The discretization of the mathematical model allows the computer simulation by using numerical algorithms.

Chapter 2

Related work

[Briefly talk about other simulation software and techniques. Also we may want to add some historical references.]

Part I

Theory

Chapter 3

Plasma introduction

Talk about what is a plasma, and why is of interest

Chapter 4

Plasma simulation

4.1 The particle-in-cell method

Solving the Vlasov equation requires a large amount of numerical resources. The particle in cell method, approximates the solution by discretization of the fields.

The method is divided in four main phases:

- Particle motion.
- Charge accumulation.
- Solve field equation.
- Interpolation of fields in particle position.

4.2 Particle mover

In order to move the particles, the equations of motion need to be solved:

$$m \frac{d\mathbf{v}}{dt} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \quad (4.1)$$

$$\frac{d\mathbf{v}}{dt} = \mathbf{v} \quad (4.2)$$

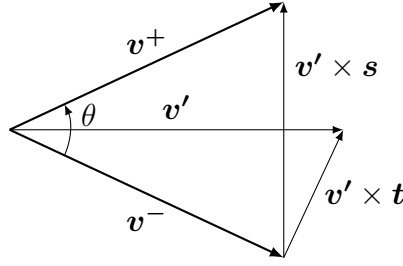
Several methods are available, but we will focus on the Boris integrator.

4.2.1 Boris integrator

Consists of three steps:

1. Add half of the electric impulse
2. Rotate
3. Add the remaining half electric impulse

The Boris integrator computes the velocity of a particle in a constant electric field \mathbf{E} and a constant magnetic field \mathbf{B} . We have the velocity $\mathbf{v}_{t-\Delta t/2}$ of the particle at $t - \Delta t/2$ as we use the leapfrog integrator.

Figure 4.1: Velocity space rotation from $\mathbf{v}-$ to $\mathbf{v}+$

Add half electric impulse We define \mathbf{v}^- as the velocity after half a electric impulse:

$$\mathbf{v}^- = \mathbf{v}_{t-\Delta t/2} + \frac{q\mathbf{E}}{m} \frac{\Delta t}{2}$$

Rotate for the magnetic field The rotation is done in two steps, first the half rotation is computed, with an angle of $\theta/2$:

$$\mathbf{v}' = \mathbf{v}^- + \mathbf{v}^- \times \mathbf{t}$$

Then the rotation is completed by symmetry, using the \mathbf{s} vector

$$\mathbf{s} = \frac{2\mathbf{t}}{1 + \mathbf{t}^2}$$

as

$$\mathbf{v}^+ = \mathbf{v}^- + \mathbf{v}' \times \mathbf{s}$$

4.3 Charge accumulation

The charge density ρ is a scalar field

4.4 Field equations

Once we have the charge density ρ we can compute the electric field \mathbf{E} by the integration of the field equations

$$\mathbf{E} = -\nabla\phi \tag{4.3}$$

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0} \tag{4.4}$$

Which can be combined into the Poisson equation

$$\nabla^2\phi = -\frac{\rho}{\epsilon_0} \tag{4.5}$$

Different methods can be used to obtain the electric field, but we will focus on matrix and spectral methods.

Chapter 5

Discrete model

The mathematical model is discretized in algebraic operations, in order to be computable.

5.1 Charge assignment

At each grid point g at \mathbf{x} we accumulate the charge of each particle p in \mathbf{x}_p as

$$\rho(\mathbf{x}) = \sum_p q W(\mathbf{x} - \mathbf{x}_p) + \rho_0 \quad (5.1)$$

The background charge density ρ_0 is used to neutralize the total charge when is non-zero. The weighting function W determines the shape of the particle charge. Different schemes can be used to approximate the charge density from the particles. We will focus on bilinear interpolation for its simplicity and low computation requirements. The corresponding weighting function can be written as

$$W(\mathbf{x}) = \begin{cases} \left(1 - \frac{|x|}{\Delta x}\right) \left(1 - \frac{|y|}{\Delta y}\right) & \text{if } -\Delta \mathbf{x} < \mathbf{x} < \Delta \mathbf{x} \\ 0 & \text{otherwise} \end{cases} \quad (5.2)$$

Notice that a particle p always affects the four enclosing grid points in the neighbourhood $\mathcal{N}(p)$, but more complex interpolation methods may extend the update region even further. It may be noted that the increase in smoothing, at computation expense, can gain from the reduced number of particles needed to obtain a similar result, avoiding nonphysical effects. The particle p has a uniform charge area, centered at the particle position \mathbf{x}_p , with size $\Delta \mathbf{x}$, as shown in the figure 5.1. Each grid point A, B, C and D receives the amount of charge weighed by the area a, b, c and d . It can be observed that the area is equal to the opposite region, when the particle p is used to divide the grid cell. The particle shape can be altered later in the Fourier space, without large computation effort, in case the solver already computes the FFT.

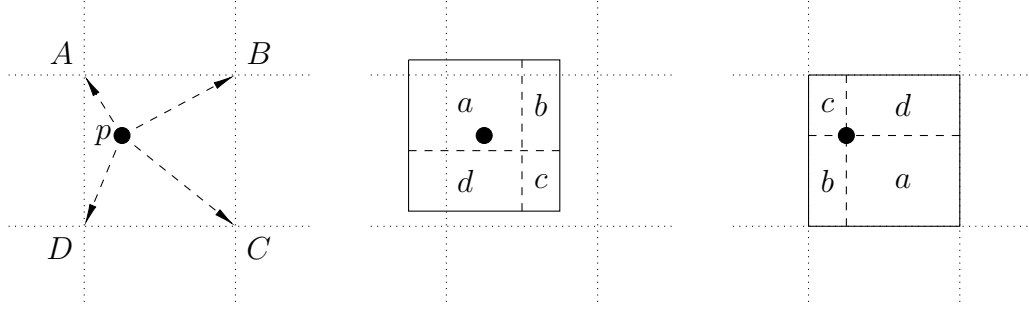


Figure 5.1: Interpolation of particle p charge into the four grid points A to D.

5.2 Field equations

In order to compute the electric field \mathbf{E} , the electric potential ϕ is generally needed, which can be obtained from the charge density ρ .

5.2.1 Electric potential

Several methods are available to solve the Poisson equation (Eq. 4.5).

Iterative methods such as Jacobi, Gauss-Seidel, Successive Over Relaxation (SOR), Chebyshev acceleration are some of the most familiar methods to solve the Poisson equation.

Matrix methods The equations from finite differencing the mesh are considered a large system of equations. We can find in this methods the Thomas Tridiagonal algorithm, Conjugate-Gradient, LU or Incomplete Decomposition.

Spectral methods Also known as Rapid Elliptic Solvers (RES) are a family of methods that use the fast Fourier transform (FFT). Are know for being usually faster than the previous ones, with a complexity in $O(N_g \log_2 N_g)$

We will only focus on the LU for small problems and for testing, and spectral methods, more specific on the Multiple Fourier Transform (MTF) method, as it is the main method implemented in the simulator, due to its relative simplicity and low computational complexity.

5.2.2 LU decomposition

For two dimensions, we can approximate the solution using the second order centered finite differences (with an error proportional to $\Delta x^2 \Delta y^2$), as

$$\frac{\phi(x-1, y) + \phi(x, y-1) - 4\phi(x, y) + \phi(x+1, y) + \phi(x, y+1)}{\Delta x^2 \Delta y^2} = -\frac{\rho(x, y)}{\epsilon_0} \quad (5.3)$$

which leads to a system of N_g linear equations and can be also written in matrix form

$$A\phi = -\frac{\Delta x^2 \Delta y^2 \rho}{\epsilon_0} \quad (5.4)$$

The $N_g \times N_g$ coefficient matrix A has non-zero coefficients only at $a_{ii} = 4$ and $a_{ij} = -1$ with $j \in \{i+1, i-1, i+N_x, i-N_x\} \bmod N_x$, for all $0 \leq i \leq N_g$. However, the matrix A is singular, so the system of equations has infinite solutions. Boundary conditions can be added to get a unique solution. The extra equation $\phi(0,0) = 0$ leads to a system with only one solution, but with one extra equation. In order to keep the matrix A square, the following steps may be taken:

1. Subtract the extra equation $\phi(0,0) = 0$ to the first row of A , with the only change in the coefficient to $a_{11} = 3$.
2. Add all first N_g equations: Each equation has one coefficient of 4 and four of -1 except the first equation. Also we assume the total charge density is zero, obtaining $\phi(0,0) = 0$.
3. Subtract it from the last equation, which leads to a zero coefficient that can be removed.

The only change that remains is at the coefficient $a_{11} = 3$. Now the matrix A is squared and non-singular and has only one solution and can now be solved with the LU method.

The LU decomposition, with a complexity in $O(2/3N_g^3)$, can be used to form two systems of equations that can be solved faster. If we rewrite the system of equations 5.4 as the usual form $Ax = b$ with

$$x = \phi, \quad b = -\frac{\Delta x^2 \Delta y^2 \rho}{\epsilon_0} \quad (5.5)$$

Then we can use the decomposition $A = LU$ to form two systems of equations

$$Ux = y, \quad Ly = b \quad (5.6)$$

which can be solved in complexity $O(2N_g^2)$.

5.2.3 Multiple Fourier Transform (MFT)

The general second-order PDE with constant coefficients and periodic boundary conditions

$$a \frac{\partial^2 \phi}{\partial x^2} + b \frac{\partial \phi}{\partial x} + c\phi + d \frac{\partial^2 \phi}{\partial y^2} + e \frac{\partial \phi}{\partial y} + f\phi = g(x, y) \quad (5.7)$$

can be solved by using the FFT. If we expand ϕ and g in a finite double Fourier series, we obtain

$$\phi(x, y) = \sum_{k,l} \hat{\phi}(k, l) \exp\left(\frac{2\pi i(xk + yl)}{n}\right) \quad (5.8)$$

and

$$g(x, y) = \sum_{k,l} \hat{g}(k, l) \exp\left(\frac{2\pi i(xk + yl)}{n}\right) \quad (5.9)$$

which now can be substituted in the Eq. 5.7, to obtain

$$\hat{\phi}(k, l) = \hat{G}(k, l) \hat{g}(k, l), \quad 0 < k < N_x, 0 < l < N_y \quad (5.10)$$

with for a unit mesh

$$\hat{G}(k, l) = \left[2a \left(\cos \frac{2\pi k}{n} - 1 \right) + ib \sin \frac{2\pi k}{n} + c + \right. \\ \left. 2d \left(\cos \frac{2\pi l}{n} - 1 \right) + ie \sin \frac{2\pi l}{n} + f \right]^{-1} \quad (5.11)$$

To solve the Poisson equation, discretized as Eq. 5.3, we have $a = d = 1$ and $b = c = e = f = 0$ so we can simplify $\hat{G}(k, l)$ as

$$\hat{G}(k, l) = \frac{1}{2} \left[\cos \frac{2\pi k}{n} + \cos \frac{2\pi l}{n} - 2 \right]^{-1} \quad (5.12)$$

Let $g = -\Delta x^2 \Delta y^2 \rho / \epsilon_0$, then the steps to compute the electric potential can be summarized as follows:

$$g \xrightarrow{\text{FFT}} \hat{g} \xrightarrow{\hat{G}} \hat{\phi} \xrightarrow{\text{IFFT}} \phi$$

1. Compute the complex FFT \hat{g} of g
2. Multiply each element of \hat{g} by the corresponding complex coefficient \hat{G} , to obtain $\hat{\phi}$
3. Compute the inverse FFT of $\hat{\phi}$ to get ϕ

The complexity in the worst case is in $O(N_g \log_2 N_g)$ with the number of total points in the grid N_g .

5.2.4 Electric field

The electric field \mathbf{E} can then be obtained by centered first order finite differences in each dimension

$$\mathbf{E}_x(x, y) = \frac{\phi(x-1, y) - \phi(x+1, y)}{2 \Delta x} \\ \mathbf{E}_y(x, y) = \frac{\phi(x, y-1) - \phi(x, y+1)}{2 \Delta y} \quad (5.13)$$

5.3 Force interpolation

The force acting on a particle p can be decomposed in two main parts, the electric and magnetic force $\mathbf{F} = \mathbf{F}_E + \mathbf{F}_B$.

The electric force \mathbf{F}_E is computed similarly as the charge deposition, but in the reverse order. The force \mathbf{F}_E is interpolated from the electric field \mathbf{E} of the neighbour grid points $\mathcal{N}(p)$, using the same interpolation function W .

$$\mathbf{F}_E = q \sum_{g \in \mathcal{N}(p)} W(\mathbf{x}_p - \mathbf{x}_g) \mathbf{E}(\mathbf{x}_g) \quad (5.14)$$

Notice that a particle p only needs the values of the electric field in the neighbourhood $\mathcal{N}(p)$.

The magnetic force \mathbf{F}_B is constant in the simulator, as we only consider a fixed background magnetic field \mathbf{B}_0 . For a particle p with velocity \mathbf{v} can be written as

$$\mathbf{F}_B = q(\mathbf{v} \times \mathbf{B}_0) \quad (5.15)$$

5.4 Equations of motion

In order to move the particles, the equations of motion need to be solved:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad (5.16)$$

$$m \frac{d\mathbf{v}}{dt} = \mathbf{F} \quad (5.17)$$

The *leap-frog* method is a common integration scheme with second-order accuracy and an error proportional to Δt^2 . The name describes the behavior of the position and velocity, which are updated at interleaved time steps, similarly to the trajectory of a frog. The method is time reversible with a stability far superior of other higher-order integration methods, such as fourth order Runge-Kutta. A more in depth stability analysis can be found in Chapter 4 of Hockney and Eastwood book [3]. The discretized equations can be written as

$$\frac{\mathbf{x}^{n+1} - \mathbf{x}^n}{\Delta x} = \mathbf{v}^{n+1/2} \quad (5.18)$$

$$m \frac{\mathbf{v}^{n+1/2} - \mathbf{v}^{n-1/2}}{\Delta x} = \mathbf{F}(\mathbf{x}^n) \quad (5.19)$$

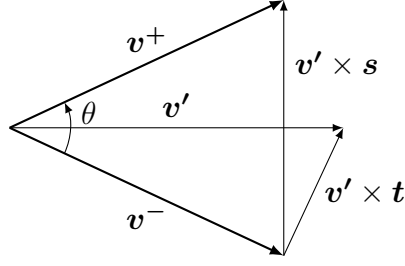
Several methods are available, but we will focus on the Boris integrator.

5.4.1 Boris integrator

Consists of three steps:

1. Add half of the electric impulse
2. Rotate
3. Add the remaining half electric impulse

The Boris integrator computes the velocity of a particle in a constant electric field \mathbf{E} and a constant magnetic field \mathbf{B} . We have the velocity $\mathbf{v}_{t-\Delta t/2}$ of the particle at $t - \Delta t/2$ as we use the leapfrog integrator.

Figure 5.2: Velocity space rotation from $\mathbf{v}-$ to $\mathbf{v}+$

Add half electric impulse We define \mathbf{v}^- as the velocity after half a electric impulse:

$$\mathbf{v}^- = \mathbf{v}_{t-\Delta t/2} + \frac{q\mathbf{E}}{m} \frac{\Delta t}{2}$$

Rotate for the magnetic field The rotation is done in two steps, first the half rotation is computed, with an angle of $\theta/2$:

$$\mathbf{v}' = \mathbf{v}^- + \mathbf{v}^- \times \mathbf{t}$$

Then the rotation is completed by symmetry, using the \mathbf{s} vector

$$\mathbf{s} = \frac{2\mathbf{t}}{1 + \mathbf{t}^2}$$

as

$$\mathbf{v}^+ = \mathbf{v}^- + \mathbf{v}' \times \mathbf{s}$$

Part II

Computation

Chapter 6

Sequential simulator

In order to begin the implementation of the simulator, an initial version was considered with the minimum complexity, to verify the correctness of the model. A graphic subsystem was build with MathGL and OpenGL to produce realtime plots of different elements of the simulation. Of special interest are the particle motion, the electric potential and the electric field.

The language of choice was C for the low overhead, the lack of automatic memory management, the support of different libraries planned in future versions and the low level design, which allowed us to define most of the data structures close to the byte level.

6.1 Design

The simulator initially only supported one group of particles of the same charge and mass, denominated specie. Each particle was implemented as a structure with a given index i , a position vector \mathbf{x} , velocity \mathbf{v} and other extra fields such as the interpolated electric field at the particle position \mathbf{E} . Only one dimension was implemented for the first tests, but soon extended to two dimensions. The fields were allocated in contiguous arrays, with the x dimension aligned with the cache line, also called row-major storage.

The configuration of the simulation is specified in plain configuration files, with the syntax defined by the `libconfig` library. Is important to allow the user to specify comments in the configuration files, as well as scientific notation in different values. Additionally, the specification of multiple species benefits from the sub-configuration block feature, which leads to a more intuitive representation. The detailed configuration is described in the chapter 13.

The solver used was initially the LU decomposition, used from the *GSL* numeric library [2], as the only focus was to obtain valid results, ignoring the performance. All implementations are tested beforehand with some test cases designed in `octave`.

6.1.1 Debug mode

In order to get insight into all the details of the simulation, a mechanism of visualization can be very useful: the different fields can be plotted in real-time for one

and two dimensions, while the particles move around. The simulator includes a visualization mode, in which the state of the simulation is plotted at a specific period of iterations (by default each iteration is shown). In this mode (which we will refer to as debug mode) the simulation is slowed down, with a top speed of 60 iterations per second, to follow the visualization in the screen.

With this mode activated, the user can observe and quickly check the overall behavior of the simulation, as is designed to minimize the delay between writing the configuration of the simulation and the execution. Once the simulator is running, the user can see several graphs being updated.

The energy measurements are always shown, including potential, kinetic and total energy—the total energy must be conserved at all times. In the case of one-dimensional simulations, the particles are plotted in the $x-v$ phase space, with the fields aligned vertically. However, in two dimensions the particles are plotted by default in the $x-y$ plane which corresponds to the physical position in space. The fields now cannot be plotted together, and only the electric potential is shown.

Once the simulation is properly tested in the debug mode, there is less chance that a misconfigured setting ruins a large simulation. This mode has also been very helpful when developing the simulator, as several tests are required to see the immediate result of a new feature, or to change a value in the configuration.

6.2 Validation

A set of different tests were designed to determine the correctness of the simulation.

6.2.1 Two particle test

A simple one-dimensional test consists of two electrons placed at some distance different of $L/2$ with no initial speed. The analytical solution is known and the motion should follow a harmonic oscillation trajectory. The energy conservation can be observed in the figure 6.1, where the total energy only varies due to the interpolation noise as the time t grows in the x axis.

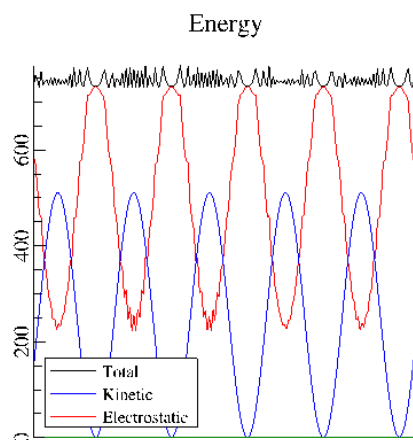


Figure 6.1: Energy conservation in two particle test as shown in the simulator (notice the lack of anti-aliasing).

6.2.2 Two stream instability

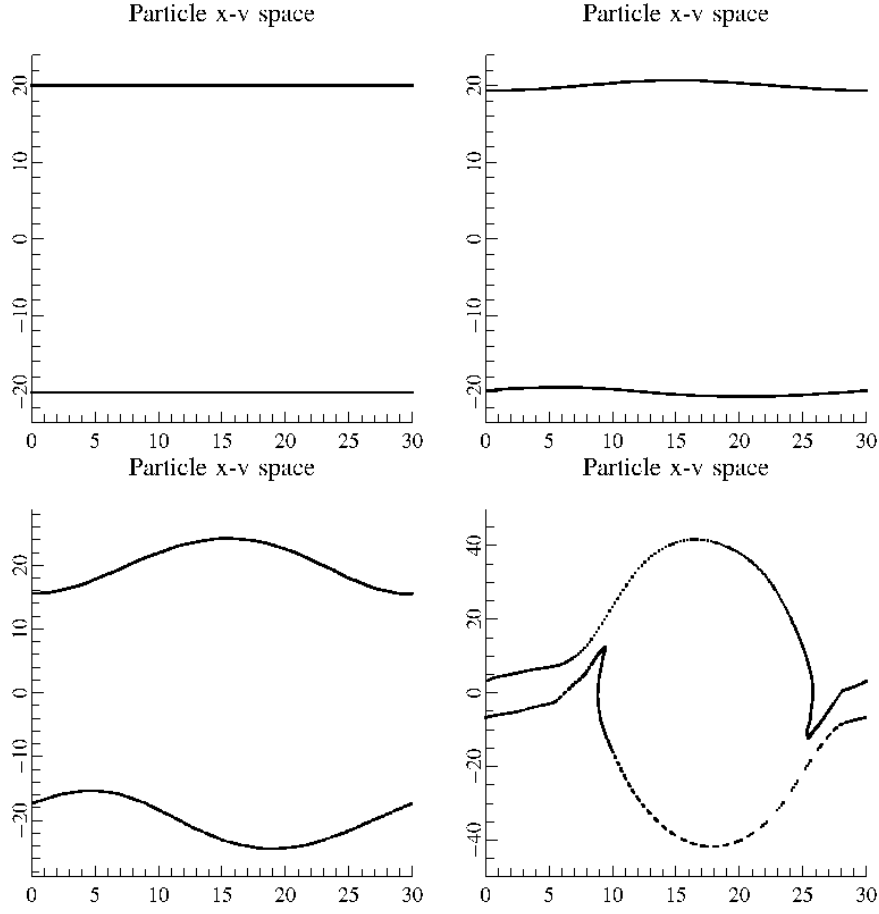


Figure 6.2: Phase space position–velocity of the two stream instability, shown at iterations: 0, 200, 400 and 600 (left to right, top to bottom)

Another example in one dimension is the two stream instability, which consists of two streams of particles with opposite velocity. With 500 particles in each stream, a very characteristic set of vortices are created in the position-velocity phase space, which can be shown in the figure 6.2.

6.2.3 Cyclotron frequency

In a simulation with two dimensions and a fixed background magnetic field \mathbf{B}_0 , a charged particle with some initial velocity should describe a circular orbit. The radius r_g known as the Larmor or gyroradius, can be computed analytically as

$$r_g = \frac{mv}{|q|B} \quad (6.1)$$

Chapter 7

Parallelization techniques

7.1 Message Passing Interface

From the need of standarize communications in a distributed computing environment, the first draft was proposed in 1992 at the Workshop on Standards for Message Passing in a Distributed Memory Environment, and has now become one of the most used communication protocol in HPC. The Message Passing Interface (MPI) provides a simple to use set of routines to allow processes distributed among different nodes to communicate efficiently.

7.1.1 Concepts

Communicator A communicator refers to a group of processes, in which each has assigned a unique identifier called the *rank*.

Point-to-point communication In order for a process to exchange information with another process, the MPI standard defines what are called point-to-point communication routines. The most common examples are `MPI_Send` to send data, and `MPI_Recv` for the reception. Both routines need the process rank of the process to stablish the connection. Additionally a tag is used to label each message, which can be specified in the reception to filter other messages.

Blocking communication The standard defines various types of communication methods for sending and receiving data. The so called blocking routines are designed such that the call does not return until the communication has been done. In the `MPI_Send` case, the call returns when the sending data can be safely modified, as has been sent or buffered. In the case of `MPI_Recv` the routine only returns when the data has been received.

Non-blocking communication Similarly as with the blocking communication, the routines `MPI_Isend` and `MPI_Irecv` don't wait until the message is sent or received to return. They return immediately, and the communication status can be checked with `MPI_Test` or the process can wait until the communication request has finished with `MPI_Wait`.

7.1.2 Implementations

7.2 OmpSs-2

OmpSs-2 is the next generation of the OmpSs programming model, composed of a set of directives and library routines. Mixes from OpenMP the annotation of source code to parallelize some sections with the StarSs execution model, based on a thread-pool design pattern.

7.2.1 Concepts

Task In OmpSs-2 a task is a section of code that can be executed independently by the runtime schedule. A task may have associated dependencies which lets the scheduler determine in wich order is allowed to execute the tasks. An example annotation of a task:

Parallelization Unless there is a unmet dependency, all tasks ready to run are executed in parallel, up to the number of CPU cores available to the runtime.

Task synchronization It may be possible that at some point in the execution all pending task are required to finish in order to continue. The directive `taskwait` allows the programmer to specify that the runtime must wait for completion of all previous created tasks.

Chapter 8

Simulator design

[After determining the mathematical model and the discretization, we want to begin the discussion on how to build the simulator.]

8.1 Field solver

Talk about MFT and the data layout.

Chapter 9

The simulator

[This chapter is focused on the simulator implementation]

9.1 Data structures

The simulator is designed for space domains of one or two dimensions. In order to parallelize the computation of each step, the space domain is distributed in blocks. First the space domain is split in one specific dimension into MPI blocks, which will be distributed among each compute node. Communications will be needed to share information between MPI blocks.

The second hierarchy splits MPI blocks into task blocks, which can run in parallel inside a compute node. Communications are not needed, as we can use shared memory in the same compute node.

Inside each task block, we have a small portion of the space domain: the grid points of the fields and the particles inside the physical space of the block. Additionally, ghost points are placed at the boundaries of the positive neighbours in each dimension of the problem.

A summary of the data layout can be seen in the figure 9.1, where the physical placement of each block corresponds to the physical position of the grid points. The 1D domain has 2 MPI blocks with 3 task blocks each, and 3 grid points per block with 1 ghost point. In total, the space is discretized in 18 grid points, which require 24 with the ghost points.

Similarly, for 2D the number of ghost points increase, as the frontier now has 2 dimensions, leading to blocks with 6 grid points and 6 ghost points. The whole domain is discretized in 120 grid points, a total of 240 with ghost points.

9.2 Simulation flow

The simulation follows a very precise set of steps to ensure the correct behavior of the physical simulation. Three main stages can be easily identified: Initialization, loop and finish.

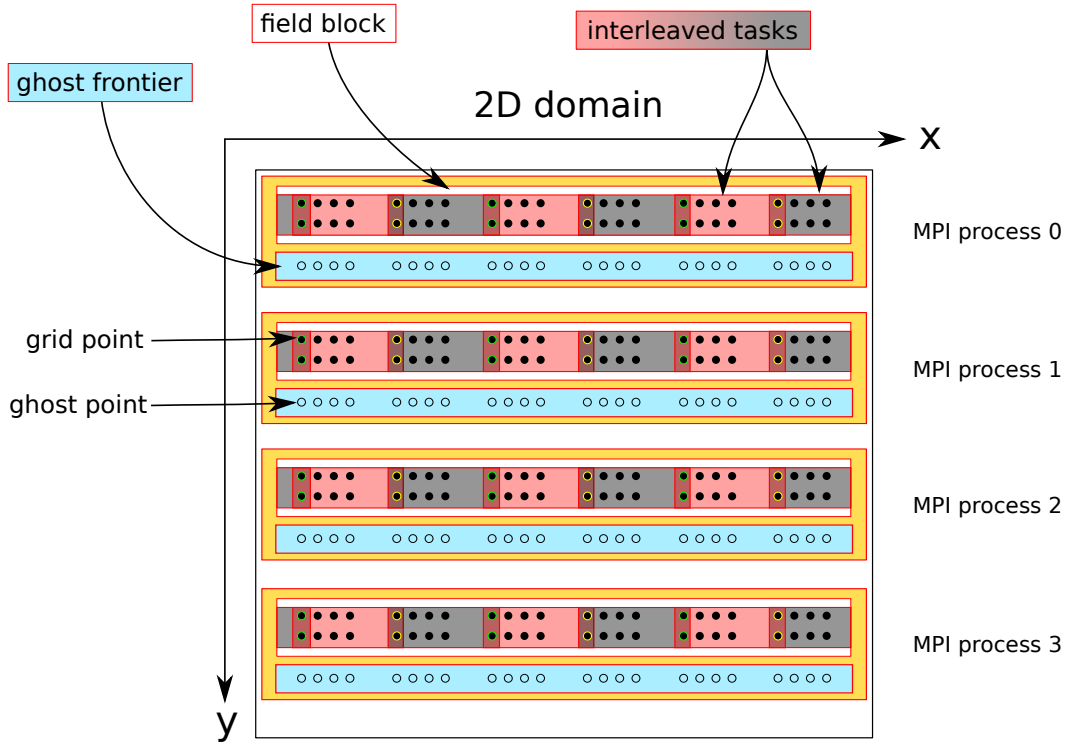


Figure 9.1: Domain blocks

9.2.1 Initialization

The iteration counter is initially set to -2 , as we are going to do two previous phases before the simulation begins:

Allocation phase After P processes were created (as in MPI processes), now we create the different structures to hold the simulation data. First the fields are distributed into task blocks, grouped in each process. For each specie, we distribute the particles based on the particle index p_i , minimizing the difference of the number of particles between blocks. The position, velocity and other parameters are set on each particle, independently of the actual block they reside.

Once all particles are initialized, we begin to move them to the correct block, based on the particle position, and increment the iteration counter. Finally, the charge density field ρ is initially computed, as we want the begin the simulation with the computation of the electric field E from ρ

Rewind phase The simulation time is not advanced equally for the speed and position of the particles. At time t the velocity computed at time $t + \Delta t/2$ whereas the position is computed at time t . In order to begin the simulation, the velocity of the particles is advanced half time-step backwards in time. This extra step is computed at the iteration $i = -1$, as we need the iteration counter to be always increasing (as is used in the message exchange as unique identifier).

9.2.2 Loop

The loop of the simulation perform four main phases:

- Solve field equation to get E from ρ .
- Interpolate E at particle positions E_p .
- Particle motion based on E_p and B_0 .
- Accumulate charge density ρ at the new position of particles.

Solver

We use the MFT method to solve the equation:

$$\nabla^2 E = -\frac{\rho}{\epsilon_0} \tag{9.1}$$

9.2.3 Finish

Here we hopefully save some information of the simulation to disk...

Chapter 10

Communication

Different communications are detailed in this chapter, such as particle and frontier communications.

10.1 Particle communication

When the particles are moved, due to the interaction with the electric field and the magnetic field, their position can exceed the boundaries of the chunk where they reside. After updating the position of each particle, the ones that exceed the chunk must be translated to the correct one. The process of particle communication is done in two stages: first the particles are moved in the X dimension, then in the Y. Several steps are required in each stage.

10.1.1 Exchange in X

All chunks in the X dimension reside in one MPI process, so the exchange of particles can be done by shared memory. Care must be taken to avoid concurrent writes in the same chunk by different tasks. The proposed solution avoids the problem by using temporal queues in each chunk. The process can be described in the following steps:

1. `collect_particles_x`: Out of bound particles in the X direction are extracted from the chunk and placed in the correct target chunk queue for local exchange.
2. `exchange_particles_x`: Each chunk looks for particles in the neighbour chunks target queues and moves them to itself.

Usually only two target queues are required for each chunk, as the particles can only move one chunk per iteration. However, in the initial iteration after the initialization of the particle positions, they can move to any other chunk, and the process is subsequently more computationally expensive. We will only focus in the general case involving only the two neighbours, as the initialization iteration can be disregarded when comparing the time against the whole simulation.

The execution order and mutual exclusion of these two phases should be guaranteed by means of a synchronization mechanism. Each step can be implemented

using OmpSs-2 tasks with dependencies, in order to exploit local parallelism. One task collects the particles out of the chunk in the corresponding queues, so it needs to access only the current chunk.

```
for(i = 0; i < plasma->nchunks; i++)
{
    chunk = &plasma->chunks[i];
    /* Place each particle outside a chunk in the X dimension, in
     * the lout list */
    #pragma oss task inout(*chunk) label(collect_particles_x)
    for(is = 0; is < sim->nspecies; is++)
    {
        collect_particles_x(sim, chunk, is, global_exchange);
    }
}
```

The execution of the corresponding exchange particle tasks will start only if the collecting step has finished in the neighbour chunks, as otherwise the queues are still being written. These dependencies must be placed in all the involved chunks.

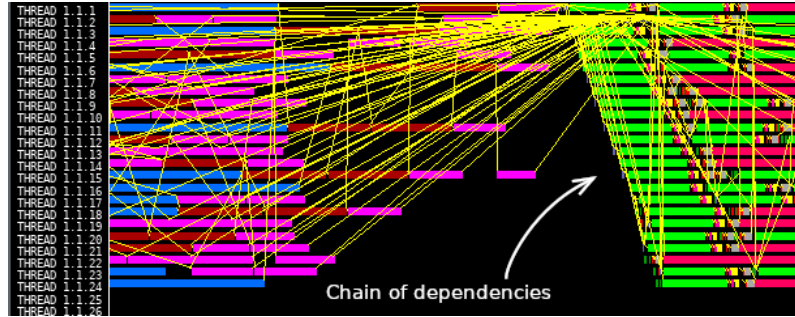
```
for (i = 0; i < plasma->nchunks; i++)
{
    chunk = &plasma->chunks[i];
    ...

    #pragma oss task inout(*chunk) \
        inout(*prev_chunk) inout(*next_chunk) \
        label(exchange_particles_x)
    {
        /* Only the two neighbours are needed */
        concat_particles(chunk, prev_chunk);
        concat_particles(chunk, next_chunk);
    }
}
```

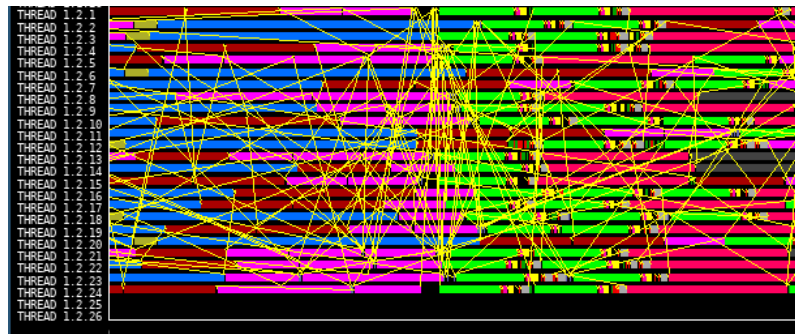
Notice that in the first iteration the exchange step must wait for all the collecting tasks to finish, as the particles can be moved to any chunk, and thus we expect to see a slower iteration than the rest of the simulation. In the following steps, only the neighbours at $i - 1$, i and $i + 1$ are required to finish the exchange process.

However, there is a problem with the previous loop: as we create the dependencies with the next chunk before the next task is created, we are building a chain of dependencies which leads to a sequential execution. Using **paraver** we can clearly see the chain in the trace graph, shown in the figure 10.1a, where no task can run in parallel until the previous one finishes. One solution to alleviate this problem is the use of a coloring technique, where each task is assigned a color. Then all tasks of the same color are created first, then the ones with the next color and so on. With three colors we ensure that the two tasks of the same color can run in parallel without concurrent access to the same chunk, as can be seen in the figure 10.2.

```
max_color = 3;
```



(a) Chain of dependencies observed



(b) The chain has been corrected

Figure 10.1: Comparison of two `paraver` traces using coloring tasks for communication.

```

for(color = 0; color < max_color; color++)
{
    /* Use coloring to prevent a chain of dependencies */
    for(i = color; i < plasma->nchunks; i+=max_color)
    {
        chunk = &plasma->chunks[i];
        ...

        #pragma oss task inout(*chunk) \
            inout(*prev_chunk) inout(*next_chunk) \
            label(collect_local_particles)
        {
            /* Only the two neighbours are needed */
            concat_particles(chunk, prev_chunk);
            concat_particles(chunk, next_chunk);
        }
    }
}

```

In the figure 10.1b it can be observed how the chain has now disappeared, and the gaps are now fully covered by tasks running in parallel.

Once all exchange tasks are completed, all particles are now placed in the correct chunk in the X dimension, and only the Y movement is left.

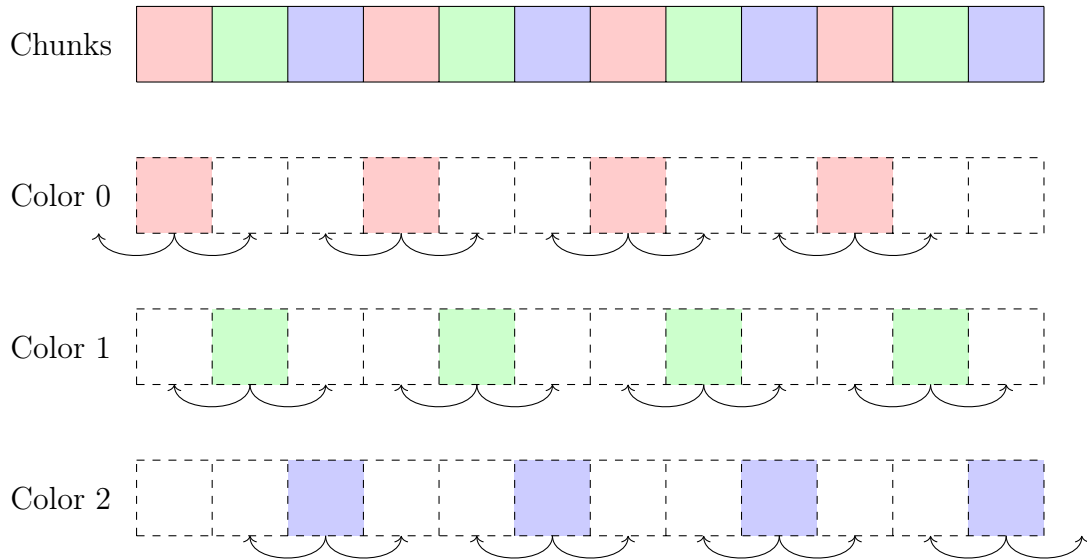


Figure 10.2: The coloring technique shown with 12 chunks were the 12 tasks are created with 3 colors.

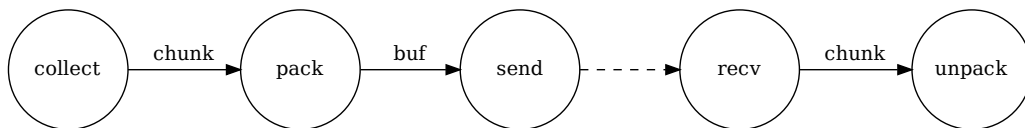


Figure 10.3: Graph of task and dependencies of particle communication in Y: Solid arrows indicate a data dependency, dashed arrows show a creation dependency order.

10.1.2 Exchange in Y

Once the particles are placed in the correct chunk in the X dimension, the displacement to the correct chunk in the Y dimension involves sending the particles to another MPI process. The steps can be resumed as

1. `collect_particles_y`: Place each particle out of the chunk bounds in a queue (one for each target destination).
2. `pack_particles_y`: Pack the particles to be sent to the neighbour chunk in a message.
3. `send_particles_y`: Send the packed particles to each neighbour.
4. `recv_particles_y`: Receive the message with the packed particles.
5. `unpack_particles_y`: Unpack the particle message and place the particles in the chunk.

Similarly as for the horizontal direction, the particles exceeding the limits of each chunk in the Y dimension are placed in a queue. Once the particles are identified

within a chunk, they are packed in a message in a contiguous memory region. This buffer is then sent using `MPI_Send` to the neighbour process.

The reception process works in the opposite order: each chunk receives the communication of the neighbour chunks in the vertical direction. Once a message is received is unpacked and the particles are added to the chunk. In the diagram 10.3 the dependencies of each step are shown in a graph.

Notice that all the MPI communication is independent of the neighbour chunks in the horizontal direction, and can be fully parallelized. Some constraints must be added to coordinate the vertical communications to guarantee that no simultaneous writes occur in the same chunk.

```
for(i = 0; i < plasma->nchunks; i++)
{
    chunk = &plasma->chunks[i];

    /* Collect particles in a queue that need to change chunk */
    #pragma oss task inout(*chunk) label(collect_particles_y)
    for(is = 0; is < sim->nspecies; is++)
    {
        collect_particles_y(sim, chunk, is, global_exchange);
    }

    /* Prepare the packet to be sent to the neighbour */
    #pragma oss task inout(*chunk) label(pack_particles_y)
    pack_particles_y(sim, chunk, i, global_exchange);

    /* Finally send the packet */
    #pragma oss task in(*chunk) label(send_particles_y)
    send_particles_y(sim, chunk, i, global_exchange);

    /* We cannot create here a task as we don't know the dependencies
     * when using MPI */
    recv_particles_y(sim, chunk, global_exchange);
}
```

10.1.3 Mitigation of deadlocks with TAMPI

Exchanging particles between processes is slightly different when using MPI or TAMPI. With the former, the design must be done with special care to avoid deadlocks: Assume each chunk tags the message with the chunk index, so the receiver can filter messages which are not from the vertical direction. Also, consider that we have multiple chunks, more than the number of CPUs available, so there are some task that cannot run in parallel and must wait.

It may happen that some task already sent messages and has reached the reception stage: it is waiting to continue and subsequently blocking the task until a message with the correct tag arrives. But other tasks may be waiting for the CPU to begin the communication and didn't send yet any message. When no CPUs are

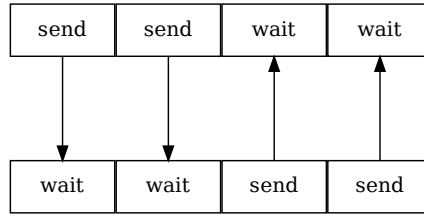


Figure 10.4: Deadlock at particle exchange in Y, where each message has a different tag.

left, a deadlock is produced as represented in the figure 10.4, where each chunk is represented by a box node and the edges show which messages were sent.

In order to palliate the deadlock using only MPI, we can avoid using the tag as a filter, so once the sending is complete, the task waits for the reception of particles from any other chunk. With this method, it is guarantee that no deadlock can occur, as before a task enters the waiting state, after sending the message, another task will be unlocked and can resume the execution.

```
#pragma oss task label(recv_particle_packet_MPI)
{
    comm_packet_t *pkt;
    MPI_Probe(MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);

    source = status.MPI_SOURCE;
    MPI_Get_count(&status, MPI_BYTE, &size);

    pkt = safe_malloc(size);

    MPI_Recv(pkt, size, MPI_BYTE, source, tag, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);

    chunk = &sim->plasma.chunks[pkt->dst_chunk[X]];

    /* We cannot create two independent tasks as we need a
     * dependence of pkt which is unknown until
     * recv_particle_packet_MPI finishes, and with chunk to
     * prevent other tasks writing in the same chunk
     * concurrently. */
    #pragma oss task inout(*pkt) out(*chunk) label(unpack_comm_packet)
    {
        unpack_comm_packet(sim, chunk, pkt);
        free(pkt);
    }
}
```

The downside of the described mechanism is the implicit complexity and the amount of extra work needed to ensure a deadlock free execution, which can be avoided with TAMPI. The deadlock is mitigated not by removing the tag, which filters the chunk, but by setting the task to sleep once it enters in the waiting state, so other tasks can begin the execution. The TAMPI library intercepts the calls to MPI and informs

the OmpSs-2 scheduler that the task can be put to sleep.

Using TAMPI only requires a minor modifications with respect to the original implementation: the message size must be known at the receiver. The current version of TAMPI doesn't include `MPI_Probe` in the family of intercepted functions. The MPI version first probes for the message to get the length and then allocates a buffer to hold the entire message. A buffer of known size may be used to hold the parts of the message while is being send. The message includes the complete size of the message in the header, so after the reception of the first message, the whole buffer can be allocated.

```
#pragma oss task out(*pkt) inout(*chunk) label(recv_particle_packet_TAMPI)
{
    ...

    size = BUFSIZE;
    pkt = safe_malloc(size);
    MPI_Recv(pkt, size, MPI_BYTE, proc, tag, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);

    /* If more data is coming, realloc and receive it */
    if(pkt->size > size)
    {
        done = size;
        size = pkt->size;
        parts_left = (size - done + (BUFSIZE - 1)) / BUFSIZE;

        /* If the packet is only a fragment, continue until we
         * fill the whole buffer */
        pkt = realloc(pkt, size);
        if(!pkt) abort();

        requests = safe_malloc(parts_left * sizeof(MPI_Request));

        /* Recv by chunks */
        for(j=0,ptr=pkt,i=BUFSIZE; i<size; j++, i+=BUFSIZE)
        {
            left = size - i;
            if(left > BUFSIZE)
                left = size;

            MPI_Irecv(ptr+i, left, MPI_BYTE, proc, tag,
                     MPI_COMM_WORLD, &requests[j]);
        }

        MPI_Waitall(parts_left, requests, MPI_STATUSES_IGNORE);
        free(requests);
    }
    ...
}
```

}

Note that all communications are done with `MPI_BYTE`, sending the packed structures as an array of bytes. This method of transmission sends the data “as-is”—MPI doesn’t perform any endianness adjustment. We assume the simulation will run within nodes with the same endianness, otherwise MPI will need information of each field or a manual process must be added before the message is unpacked. Additionally, the structures sent over MPI are packed to avoid any holes in the buffer sent.

10.2 Field communication

Each MPI process holds a block with the different fields of the assigned region of space. Due to the interpolation process some elements of the neighbour fields are needed to complete the interpolation, which implies that additional communication is needed.

10.2.1 Charge density ρ

Following the order of the simulation, first the ρ field is updated, where all particles deposit the charge. Given a ρ field of size (n_x, n_y) an extra row ρ_{n_y} is added to hold the first row of ρ of the next block ρ_0 , as shown in the figure 10.5. Notice that an extra padding is required by the FFTW library, to accommodate intermediate results, and must be taken into account when designing the communications.

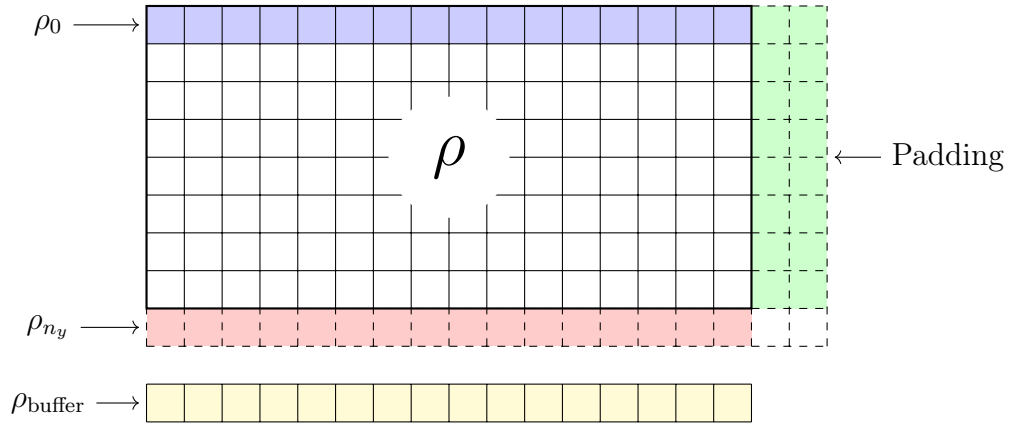


Figure 10.5: The field ρ with the padding

Each process sends ρ_{n_y} to the next process, and receives ρ_0 from the previous one. The size of the message is constant and known beforehand, so it can be stored in the same buffer ρ_{buffer} in each iteration, which is added to ρ_0 to finally complete ρ where the ghost is no longer needed.

If the ghost is sent before we begin the reception process, we can reuse ρ_{n_y} to hold also ρ_{buffer} . But the communications use non-blocking communications, which may not finish when the process reaches the reception step, so an additional buffer is used instead—a technique known as double buffering. Before each send the status

of the previous request is tested, and in case is not yet finished, we wait before continue. The two main functions `comm_mat_send` and `comm_mat_recv` are used to transfer a buffer of floating point numbers.

```
int comm_mat_send(sim_t *sim, double *data, int size, int dst,
                 int op, int dir, MPI_Request *req)
{
    int tag = compute_tag(op, sim->iter, dir, COMM_TAG_DIR_SIZE);

    if(*req)
        MPI_Wait(req, MPI_STATUS_IGNORE);

    return MPI_Isend(data, size, MPI_DOUBLE, dst, tag,
                    MPI_COMM_WORLD, req);
}

int comm_mat_recv(sim_t *sim, double *data, int size, int dst,
                 int op, int dir)
{
    int tag = compute_tag(op, sim->iter, dir, COMM_TAG_DIR_SIZE);

    return MPI_Recv(data, size, MPI_DOUBLE, dst, tag,
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

10.2.2 Electric potential ϕ

After the solver, the result is stored in a similar way as ρ , with padding at the right. However, now we have more ghosts at the top and at the bottom of ϕ , as they will be needed to compute the electric field \mathbf{E} . In the figure 10.6 the different regions can be seen: In blue the ones which will be send, ϕ_0 and ϕ_1 to the previous process to fill φ_{n_y} and φ_{n_y+1} , and ϕ_{n_y-1} to be sent to the next process, to fill φ_{-1} . Notice

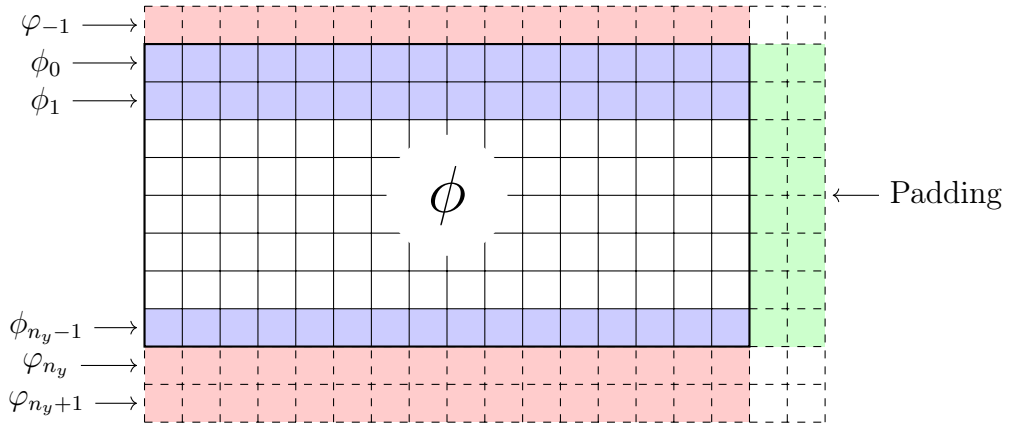


Figure 10.6: The electric potential ϕ with the ghost rows (red) and padding (green)

the use of the notation φ to denote the ghosts and ϕ the rows of the actual field.

It can be observed that the first two rows ϕ_0 and ϕ_1 are not consecutive in memory, as they have the padding at the right. To avoid two messages or additional copies, the two rows are sent with the padding included, which will be placed “as-is” in the receiving process at φ_{n_y} and φ_{n_y+1} , as the padding region is ignored.

10.2.3 Electric field \mathbf{E}

The electric field \mathbf{E} can be computed from the ghosts of the electric potential ϕ without the need of extra communications. The electric field E_x with a periodic boundary is obtained from ϕ , as the whole space domain is available in the block in the X dimension. In the case of E_y we will need the ghost row at n_y , which is marked

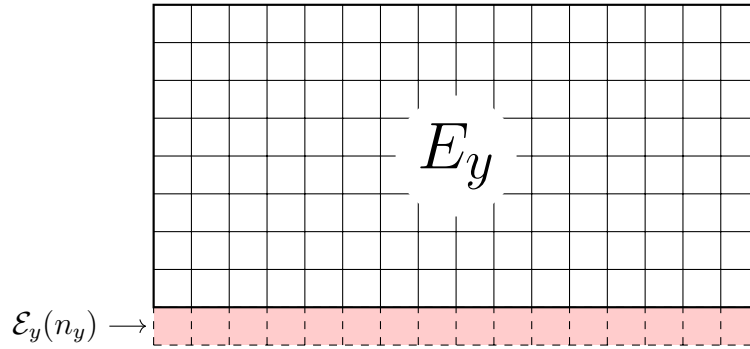


Figure 10.7: The electric field E_y with the ghost row at n_y (red)

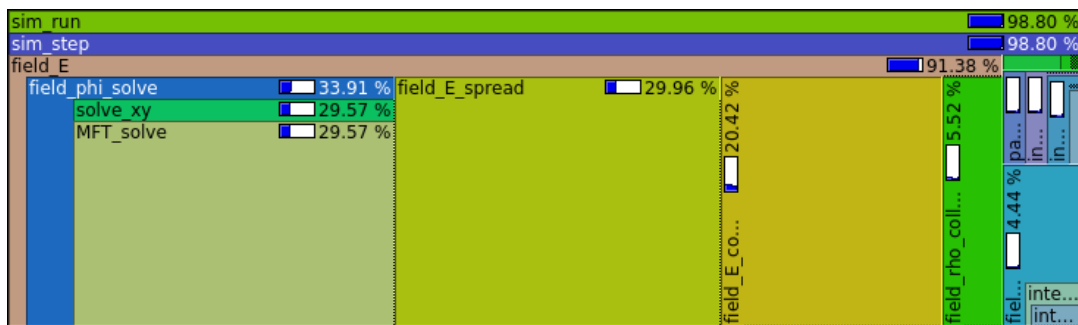
in red in the figure 10.7, in order to interpolate the electric field in the particles of the block. But the computation of the whole field can be produced from the extra ghost rows stored in ϕ , which were precisely placed to avoid another communication step.

Chapter 11

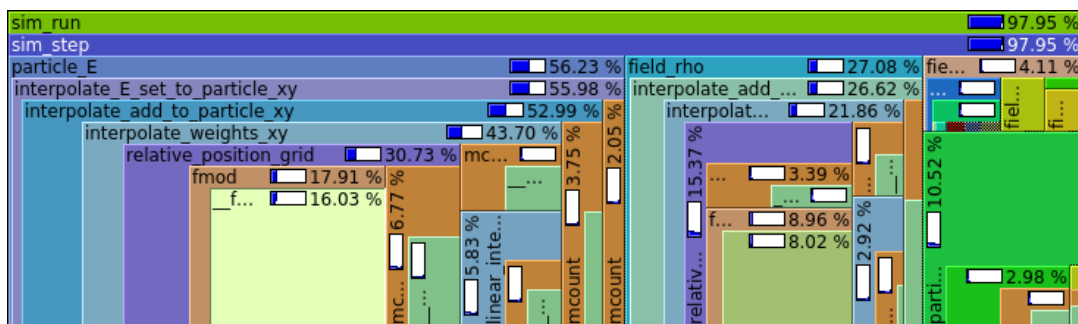
Analysis

In order to reduce the amount of CPU time involved in each step of the simulation, the best strategy is to reduce the time spent in the most time consuming part.

The CPU time involved in each part of the simulation may depend on various factors, such as the number of grid points, the number of particles or the boundary conditions. As an example, consider a simulation with a large number of grid points, with few particles—the computation of the electric field (`field_E`) will dominate the simulation time, as shown in the figure 11.1a. In a case of a large number of particles and a smaller grid, the particle interpolation (`particle_E`) dominates the whole execution as seen in the figure 11.1b. In order to optimize the general use case,



(a) 1024 particles, 512x512 grid points



(b) 10240 particles, 64x64 grid points

Figure 11.1: Comparison of the time spent in each function at two different simulations.

different inputs will be tested and the main simulation steps will be characterized. Furthermore, different algorithms or methods may be used to improve the speed.

As an example, the LU algorithm is compared with the spectral method MFT.

11.1 Analysis with varying inputs

Chapter 12

Analysis of performance

The execution time is measured and analyzed in detail.

12.1 Variables of interest

A simple test to determine the performance of the simulator is to measure the time per iteration (using the wall clock). The time spent in the different stages of the simulation can also be of interest.

Consider the sequence of time per iteration (also referred simply as the time) T_0, \dots, T_n as independent random variables from a common distribution with an unknown mean μ and finite standard deviation σ . The sample mean \bar{T} can be approximated with a certain degree of confidence by a process of sampling.

Different parameters of the simulation may affect the time mean μ and is the objective of this chapter to find out what the relation is. The time is at least in $O(N_p)$, as we need to iterate over each particle N_p . We also know that the worst-case complexity of the FFT is in $O(N_g \log N_g)$ for N_g total grid points.

The simulator runs for at least 30 iterations, and then the standard error of the mean (SEM) is computed. Then in each iteration is determined if the SEM is below a threshold to stop the simulation. The threshold is determined so that when the simulation stops, the mean is has an error of at most 0.2% with probability 95%.

12.1.1 Number of particles

The number of particles N_p is one of the main parameters that affect the running time of each iteration. In order to characterize the scalability of the application, several tests are designed with increasing number of particles, and different metrics are compared.

The wall clock is used in the process with rank zero to measure how long takes each iteration.

12.1.2 Constant number of CPUs

The simulator is designed to scale with the number of particles when the number of CPUs or process are incremented, as each chunk can be computed in parallel. But

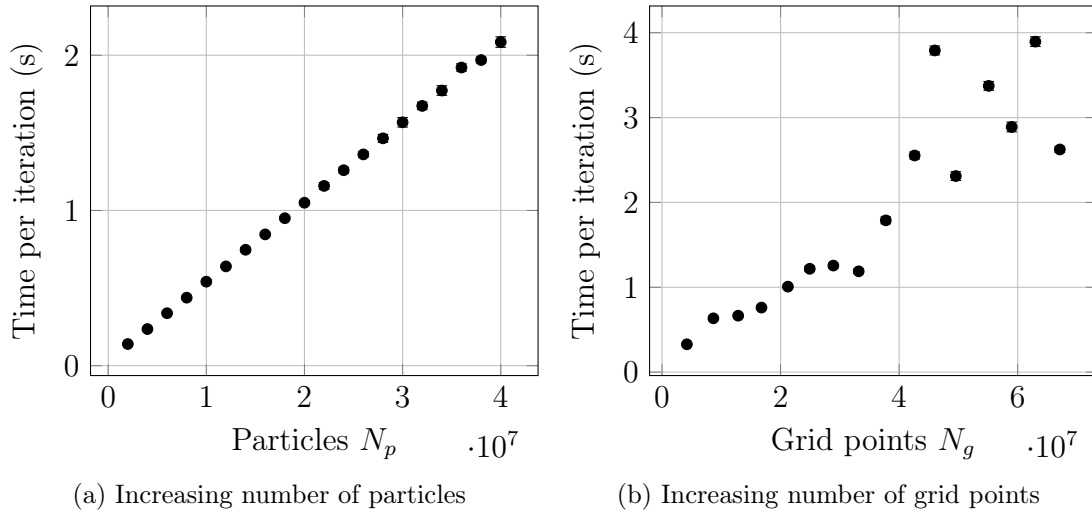


Figure 12.1: The effect of the variables N_p and N_g to the time per iteration. Using one process and 32 CPUs (MPI communications are not needed).

when the number of grid points is incremented, the FFT solver must scale both in the number of CPUs and processes. The FFTW library offers two parallelization implementations for multithreading: Using OpenMP and POSIX threads (pthreads). OpenMP is not compatible with OmpSs-2 as we have one runtime already running so the pthread implementation was tested. Unfortunately, the FFTW library doesn't

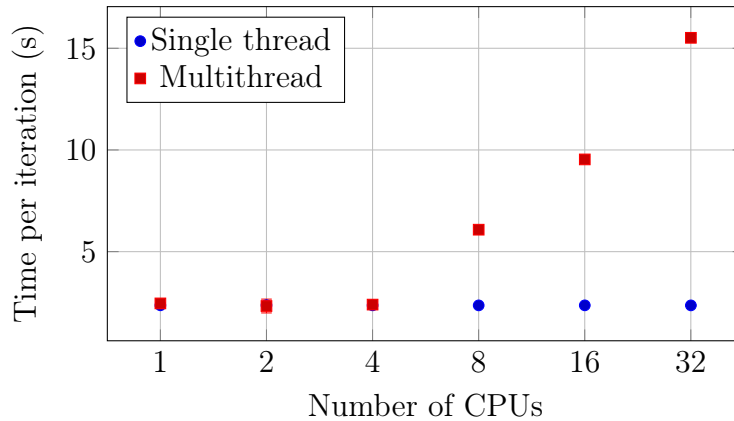


Figure 12.2: The number of CPUs is increased with only one process: the solver cannot scale and the time per iteration increases. Configuration used: $N_p = 5 \times 10^5$, $N_g = 8192 \times 8192$.

show a good speedup, in fact worsens the time per iteration when adding more threads with the configurations tested. In the figure 12.2 it can be shown how the time grows as the number of CPUs increases. The FFTW documentation warns about this problem, claiming that it can only improve the time with large enough matrices:

A shared-memory machine is one in which all CPUs can directly access the same main memory, and such machines are now common due to the ubiquity of multi-core CPUs. FFTW's multi-threading support

allows you to utilize these additional CPUs transparently from a single program. However, this does not necessarily translate into performance gains—when multiple threads/CPUs are employed, there is an overhead required for synchronization that may outweigh the computational parallelism. Therefore, you can only benefit from threads if your problem is sufficiently large.

However, larger matrices are not useful to get more precise results, as we rather prefer an increase in the number of particles than in grid points.

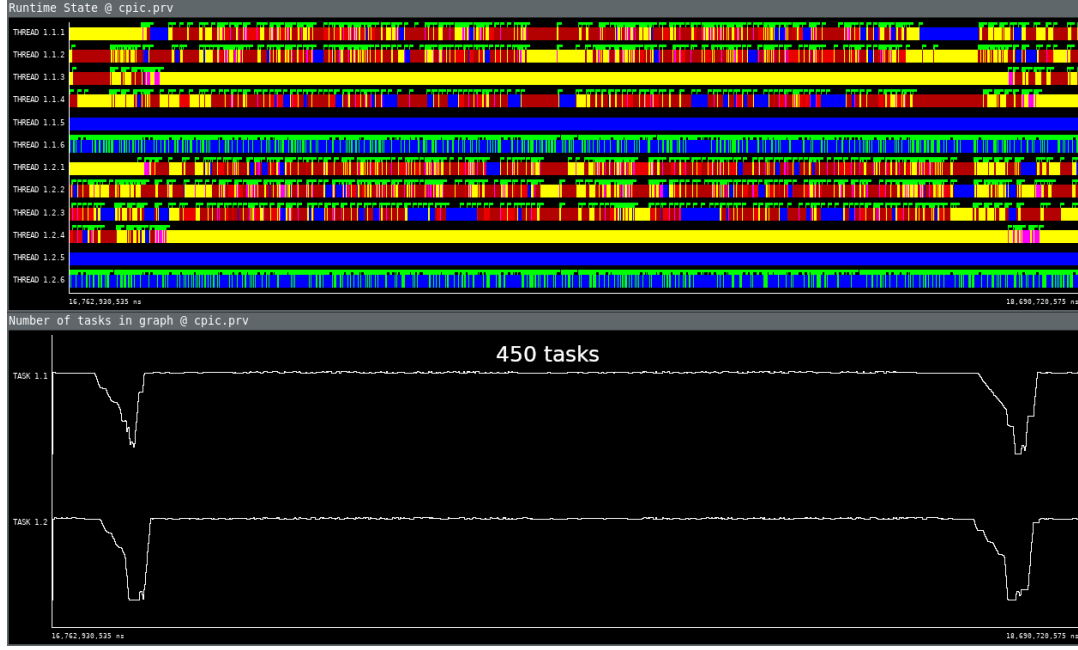


Figure 12.3: Tasks created inside the FFTW when using OmpSs-2: Up to 450 tasks are created in rapid succession, with only 4 CPUs and 2 processes.

In order to avoid a scalability problem, another approach was tested: Add support for OmpSs-2 in the FFTW to enable multithreading, following the same structure as OpenMP. The results obtained were similar as with the pthread case, but more insight was gained in how the task were created. It shown that the overhead added by the large amount of created and destructed quick tasks outweigh any benefit that could be gained by multithreading.

We can mitigate the effect of the scaling by increasing the number of processes. In order to evaluate which ratio of processes and CPUs yields the best performance several configurations are tested. With a fixed number of maximum CPUs available set to 32, we increase the number of processes while we reduce the CPUs per process.

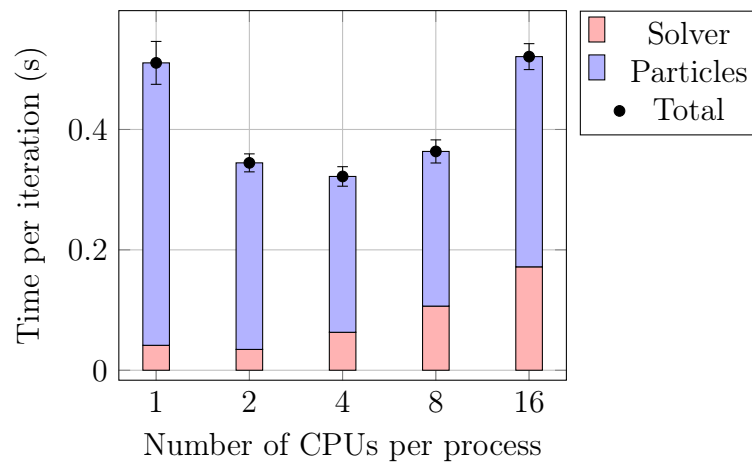


Figure 12.4: The number of CPUs per process is incremented while reducing the number of processes (the total number of CPUs is set to 32 and is kept constant). The time per iteration is measured, which leads to a characteristic U shape.

Chapter 13

Configuration

Chapter 14

Caveats and limitations

Here is a list of things that are not implemented in the simulator, but may be added in a future work.

- More than 2 dimensions
- Fully electromagnetic simulation
- Relativistic particle movement
- Heterogeneous architecture (GPU+CPU...)
- Energy conserving codes
- Visualization of big simulations (paraview)
- Replace simulation units, so we avoid factor multiplications
- Other interpolation methods (reduce noise at computational cost)

Caveats that need to be fixed

- Allow to specify plasma frequency
- Validation with other simulation codes

Bibliography

- [1] F. F. CHEN, *Introduction to Plasma Physics and Controlled Fusion: Volume 1: Plasma Physics*, Plenum Press.
- [2] M. GALASSI, J. DAVIES, J. THEILER, B. GOUGH, G. JUNGMAN, P. ALKEN, M. BOOTH, AND F. ROSSI, *GNU Scientific Library Reference Manual*, Network Theory Ltd., third ed., 2009.
- [3] R. W. HOCKNEY AND J. W. EASTWOOD, *Computer Simulation Using Particles*, Taylor & Francis, Inc., Bristol, PA, USA, 1988.