Particle-in-cell plasma simulation with OmpSs-2

Place proper cover page

Rodrigo Arias Mallo

June 26, 2019

Contents

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

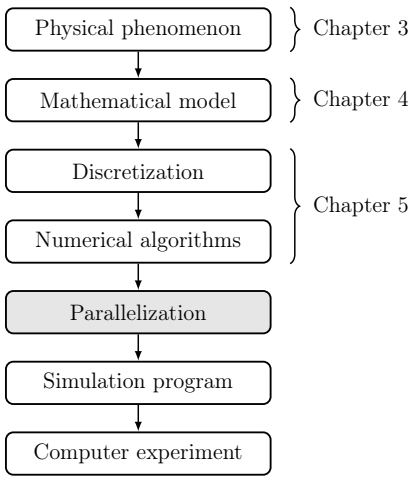


Figure 1.1: Principal steps in computer experiment

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 Also we may want to add some historical references.]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

3.1 Everyday plasmas

It may be surprising to find out that when we look at the universe, the most common state of matter is plasma, which is a ionized gas formed by free electrons and ions at a region in space–often known as the fourth state of matter. The sun, our closest star, is a giant ball of plasma

Most common forms of plasma only occur in vaccum, as otherwise the air cools the plasma and returns to a gas.

In our planet, we can see forms of plasma almost every day. A storm day the lightnings. The spark on some piezoelectric lighters, which is the very same principle that occurs in gasoline engines.

The Aurora Borealis, of the lightning of a fluorescent tube or the pixels of a plasma TV.

A precise definition of a plasma is given by Chen [1] as *"a quasineutral gas of charged and neutral particles which exhibits collective behavior"*. The

Chapter 4

Plasma simulation

4.1 The particle-in-cell method

Solving the Vaslov equation requires a large amount of numerical resources. The particle in cell method, approximates the solution by discretization of the fields and by interpolation of the grid to the particles. The method is divided in four main phases

- **Charge accumulation**: The charge density is interpolated in the grid from the particle positions.
- **Solve field equation**: From the charge density ρ the electric potential is obtained ϕ and then the electric field $\boldsymbol{E}$.
- **Interpolation of electric field**: The electric field is interpolated back to the particle positions.
- **Particle motion**: The force is computed from the electric field at the particle position and the particle is moved accordingly.

Complete the description of the method

4.2 Particle mover

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

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

$$\frac{d\boldsymbol{v}}{dt} = \boldsymbol{v} \tag{4.2}$$

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

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 $\boldsymbol{E}$ by the integration of the field equations

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

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

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 $\boldsymbol{x}$ we accumulate the charge of each particle p in $\boldsymbol{x}_p$ as

$$\rho(\boldsymbol{x}) = \sum_p q\, W(\boldsymbol{x} - \boldsymbol{x}_p) + \rho_0 \tag{5.1}$$

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 it's simplicity and low computation requirements. The corresponding weighting function can be written as

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

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 $\boldsymbol{x}_p$, with size $\Delta\boldsymbol{x}$, 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.

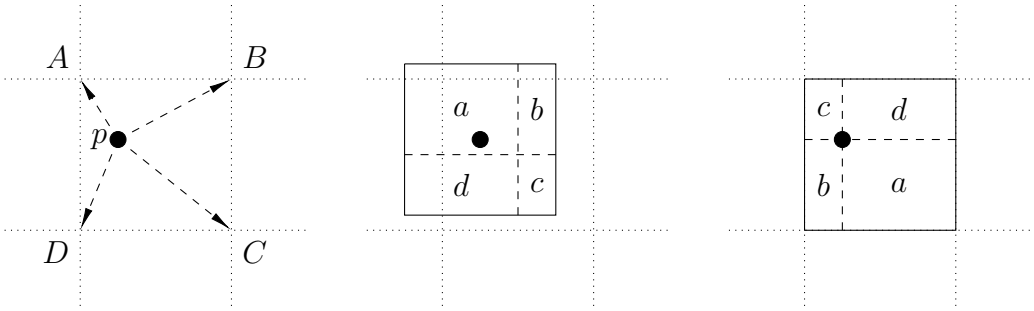


Figure 5.1: Interpolation of particle p charge into the four grid points A to D.

5.2 Field equations

In order to compute the electric field $\boldsymbol{E}$, 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 $j \in \{i+1, i-1, i+N_x, i-N_x\} \mod N_x$, for all $0 \leq i \leq Ng$. 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

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

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

$$Ux = y, \quad Ly = b \tag{5.6}$$

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

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

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

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

and

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

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

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

with for a unit mesh

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

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

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

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 $\boldsymbol{E}$ can then be obtained by centered first order finite differences in each dimension

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

5.3 Force interpolation

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

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

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

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

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

$$\boldsymbol{F}_B = q(\boldsymbol{v} \times \boldsymbol{B}_0) \tag{5.15}$$

5.4 Equations of motion

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

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{v} \tag{5.16}$$

$$m\frac{d\boldsymbol{v}}{dt} = \boldsymbol{F} \tag{5.17}$$

The *leap-frog* method is a common integration scheme with second-order accuracy and an error proportional to Δt^2. The name describes de 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 an 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

$$\frac{\boldsymbol{x}^{n+1} - \boldsymbol{x}^n}{\Delta \boldsymbol{x}} = \boldsymbol{v}^{n+1/2} \tag{5.18}$$

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

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

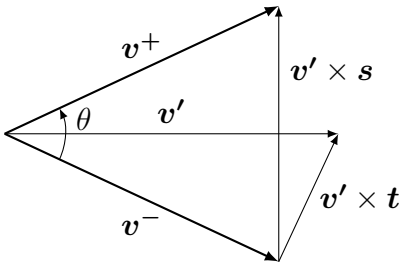


Figure 5.2: Velocity space rotation from $\boldsymbol{v}-$ to $\boldsymbol{v}+$

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

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

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

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

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

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

as

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

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 $\boldsymbol{x}$, velocity $\boldsymbol{v}$ and other extra fields such as the interpolated electric field at the particle position $\boldsymbol{E}$. 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 **??**.

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`.

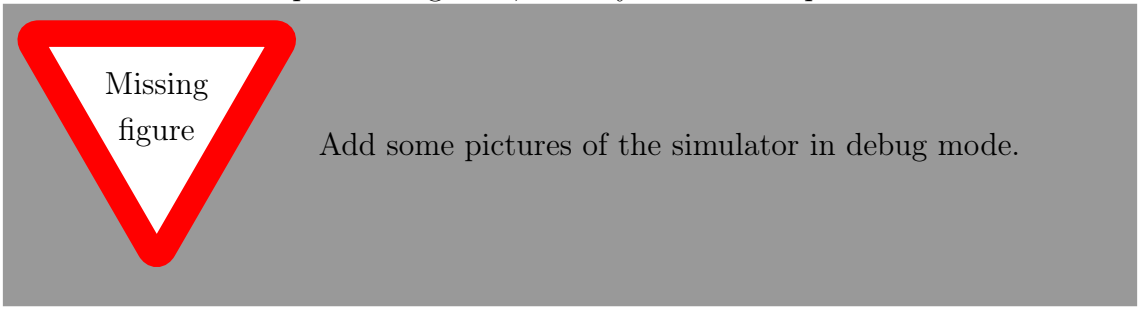
6.1.1 Debug mode

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, an 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 being very helpful when developing the simulator, as several test 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.

6.2.2 Two stream instability

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.

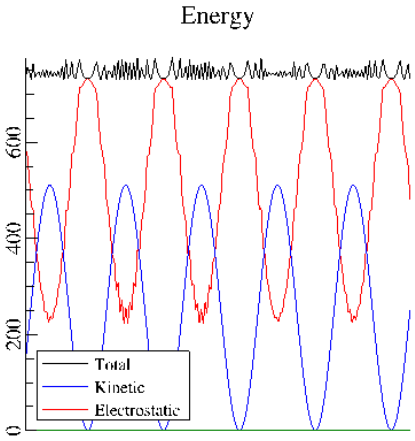


Figure 6.1: Energy conservation in two particle test as shown in the simulator (notice the lack of anti-aliasing).

6.2.3 Cyclotron frequency

In a simulation with two dimensions and a fixed background magnetic field $\boldsymbol{B}_0$, 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

$$r_g = \frac{mv}{|q|B} \tag{6.1}$$

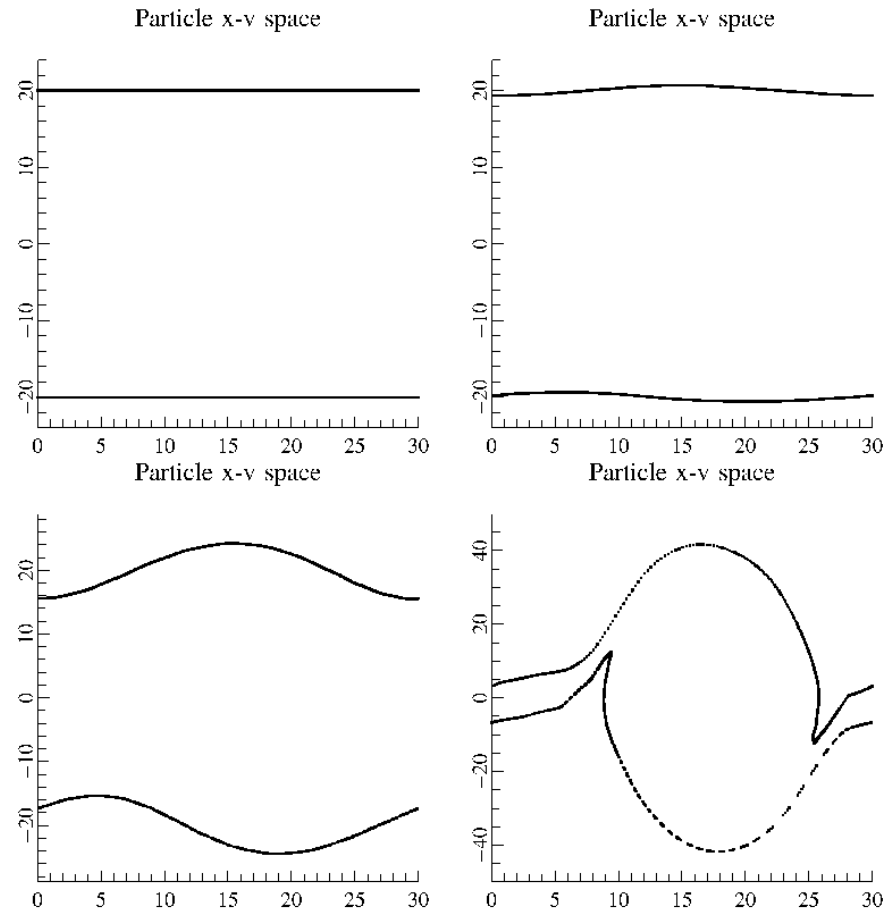


Figure 6.2: Phase space position–velocity of the two stream instability, shown at iterations: 0, 200, 400 and 600 (left to right, top to bottom)

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 comunicate 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 inmediately, 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. The notation used to describe a task is by the utilization of the `#pragma` directive, por example:

```
#pragma oss task inout(a[0:N-1]) in(b[0:N-1])
for(i=0; i < N; i++)
      a[i] += b[i];
```

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 syncronization 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

9.1 Decomposition

To parallelize the simulation, the process must be decomposed in parts that can be executed in parallel and several decompositions are known. One of the most common technique found in particle-in-cell codes, is the *domain decomposition*—the physical space is divided into sections of similar size and the fields are assigned to different computing units. The main drawback of this technique is the risk of unbalanced load, as some regions of space may contain a large amount or even all the particles.

Another approach called *particle decomposition* consists in the division of the particles into groups, where each processor maintains a copy of the fields of the whole space. The problem of this method is the limitation of scalability, as the number of grid points used in the fields is limited by the memory of one computing element.

Additionally, the Fourier transform needed by the MFT solver is implemented using the FFTW library and the parallelization design provided by the library introduces a constraint in the distribution of the fields: they need to be broken into slices in the Y dimension, resulting in contiguous blocks of elements in X. Consequently, the domain decomposition is the chosen technique for the simulator.

Firstly, the space domain is distributed in blocks by splitting the physical space in the Y dimension, as shown in the figure 9.1, and each block is assigned to an MPI process. As the simulation evolves, communications are needed to exchange information between processes. The particles enclosed within a block also are assigned to the same process in order to speed up the interpolation process. Furthermore, a second hierarchy splits the particles of a process into plasma chunks, which can be processed in parallel. In this case communications within the chunks of a process are not needed as we can use shared memory to exchange information. Notice that the number of chunks can vary to fit the number of CPUs.

We will refer to a block to denote the region of space assigned to a process and the grid points contained in that region. On the other hand a chunk has also a region of space assigned of a block, but always is associated with a group of particles.

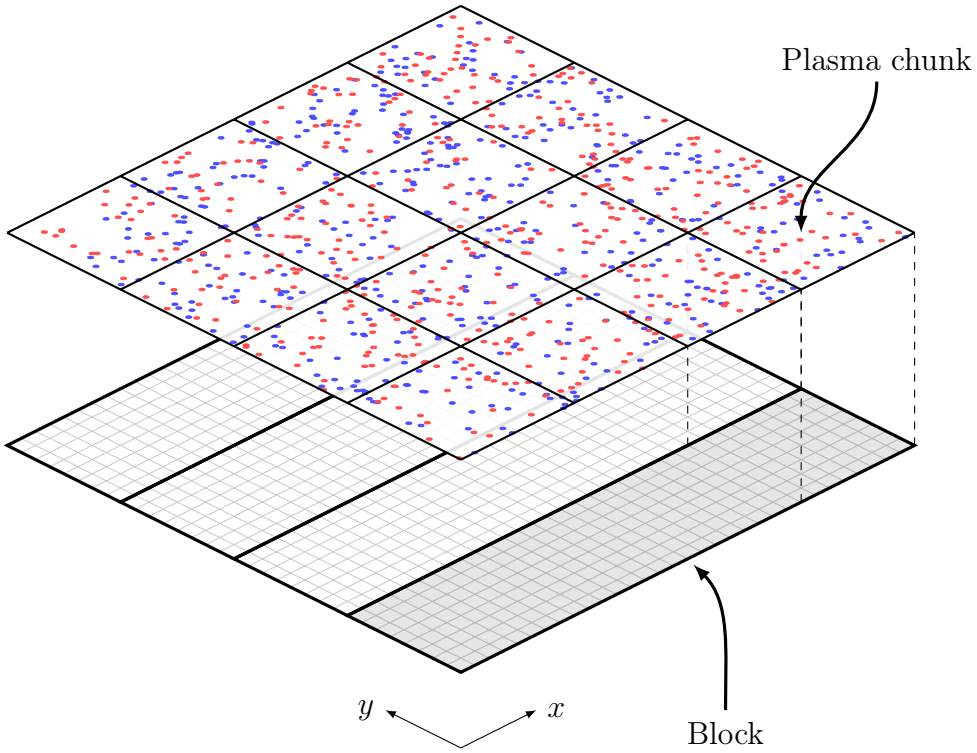


Figure 9.1: Domain decomposition: The plasma is divided into chunks in both directions and the fields into blocks in the Y dimension only

9.2 Data layout

Each block contains the three fields needed for the simulation: the charge density ρ, the electric potential ϕ and the electric field $\boldsymbol{E}$, which can be decomposed in the two components E_x and E_y. As a consequence, a total of four matrices are needed to store the three fields.

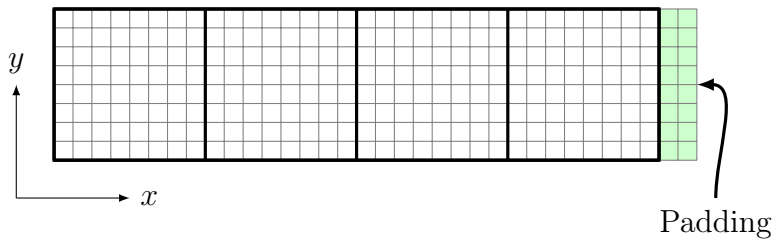


Figure 9.2: A block divided in four regions, each corresponding to a plasma chunk. Extra padding is added at the right for internal use in the FFTW library.

A simplified representation of a block can be observed in the figure 9.2, where the X dimension of each field is contiguous in memory. Notice the padding region in green, which is needed for the FFTW library to store intermediate values. The use of ghost elements is needed for communications and will be detailed in the chapter 10. If we look at each cell (x, y) in the block we find the four components $\rho(x, y)$, $\phi(x, y)$, $E_x(x, y)$ and $E_y(x, y)$.

9.3 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.

9.3.1 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).

9.3.2 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:

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

9.3.3 Finish

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

9.4 Parallelization

9.4.1 Charge accumulation

The interpolation process described in the equation **??** is executed in parallel for all the particles of each chunk. The charge density field is being updated in parallel, which involves the four surrounding grid points of a particle, and it may happen that at the frontier of two chunks a concurrent access to the same element occurs.

To avoid a race condition, a set of tasks are defined with OmpSs-2 with dependencies with the conflicting chunks.

```
for (i=0; i<plasma->nchunks; i++)
{
      c0 = &plasma->chunks[i];
      c1 = &plasma->chunks[(i + 1) % plasma->nchunks];
      #pragma oss task commutative(*c0, *c1) label(rho_update_0)
      rho_update(sim, i);
}
```

Listing 9.1: Task to update ρ field using the `commutative` directive

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.

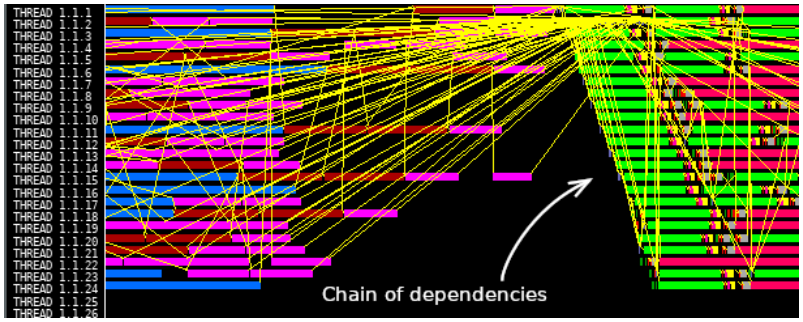
```
for(i = 0; i < plasma->nchunks; i++)
{
      chunk = &plasma->chunks[i];
      /* Place each particle outside a chunk in the X dimension, in
       * the lout list */
      #pragma oss task inout(*chunk) label(collect_particles_x)
      for(is = 0; is < sim->nspecies; is++)
      {
            collect_particles_x(sim,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

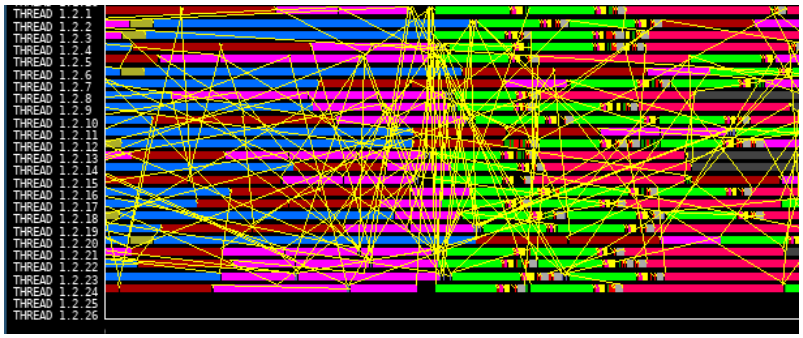
```
for (i = 0; i < plasma->nchunks; i++)
{
      chunk = &plasma->chunks[i];
      ...

      #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.



(a) Chain of dependencies observed



(b) The chain has been corrected

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

```
max_color = 3;

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.

This technique can be expressed without extra work, by using the directive

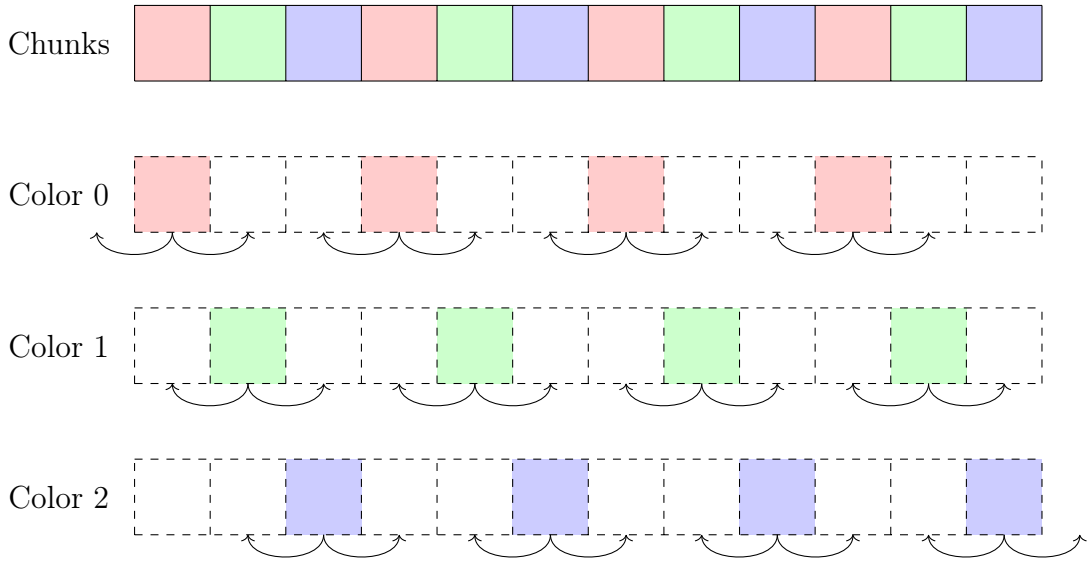


Figure 10.2: The coloring technique shown with 12 chunks were the 12 tasks are created with 3 colors.

`commutative`, which acts similarly as `inout` but can be executed in any order. Then, once a task begins execution, locking the next chunk, other unordered chunk can be executed in any order, if their neighbour chunks are unlocked.

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

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

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.

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).

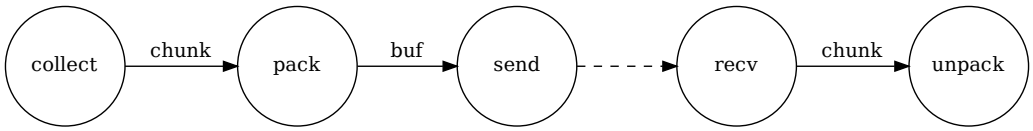


Figure 10.3: Graph of task and dependencies of particle communication in Y: Solid arrows indicate a data dependency, dashed arrows show a creation dependency order.

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 */
```

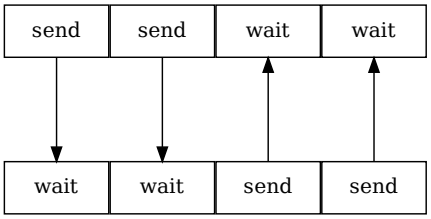


Figure 10.4: Deadlock at particle exchange in Y, where each message has a different tag.

```
    #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 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 inout(*chunk) weakinout(chunk[0:Nc-1]) \
    commutative(*sim) 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);

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

        #pragma oss task inout(*pkt) inout(*recv_chunk) label(
            unpack_comm_packet)
        {
              unpack_comm_packet(sim, recv_chunk, pkt);
              free(pkt);
        }
}
```

It must be enforced that the calls to `MPI_Recv` and `MPI_Probe` are done in mutual exclusion, as otherwise another task could receive the message between the tho calls. The dependency with the sentinel `*sim` avoids the problem, which is released as soon as the packet is received. Notice that, in order to create a nested task to process the chunk stated in the packet, we must indicate in the `weakinout` directive all posible chunks that may be selected. Then, only one will be used by the child task to unpack the particles.

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 comming, 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);
        }
        unpack_comm_packet(sim, chunk, pkt);
        free(pkt);
}
```

Note that all communications are done with `MPI_BYTE`, sending the packed structures as an array of bytes. This methods 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.

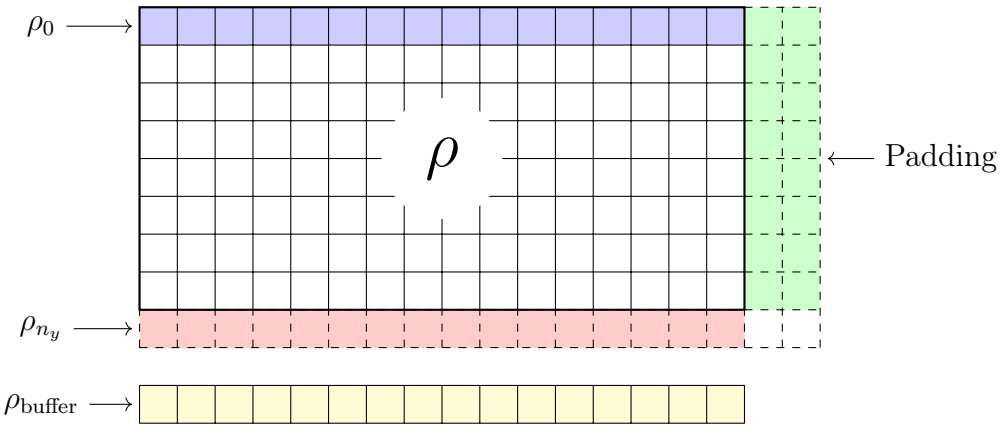


Figure 10.5: The field ρ with the padding

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 the 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 $\boldsymbol{E}$. 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

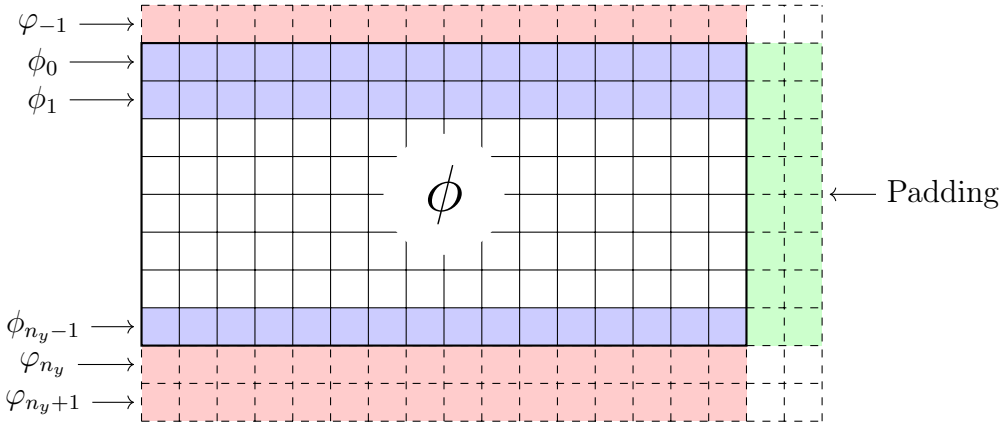


Figure 10.6: The electric potential ϕ with the ghost rows (red) and padding (green)

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 $\boldsymbol{E}$

The electric field $\boldsymbol{E}$ 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 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.

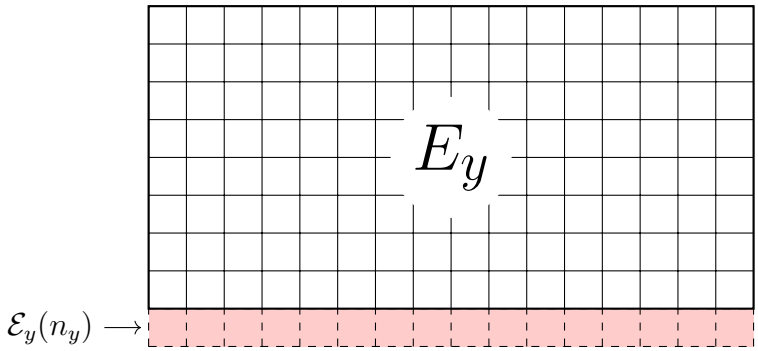


Figure 10.7: The electric field E_y with the ghost row at n_y (red)

Chapter 11

Analysis of performance

The time of the simulation will be used to characterize the performance when several parameters are changed. The time is measured using the wall clock, and refers to the *time per iteration*. Sometimes the different stages of the simulator will be measured as well, to give more insight in the distribution of the time. Notice that each process may start or end a iteration at different times, by in the long run all processes must be synchronized. The measurements will take place only in the first process (with rank zero).

At each iteration various factors may affect the iteration time and introduce a random delay. We will model the simulation time as $t = \hat{t} + e_t$, where the true simulation time $\hat{t}$ is unknown but constant between iterations, and the error e_t is a random variable with zero mean and unknown but finite variance σ^2. Additionally, we will assume that the error e_t is independent and identically distributed in each iteration of the same configuration and that follows a normal distribution.

We can then consider the sequence of measured times $T = t_1, \ldots, t_n$ as independent random variables from a common distribution with an unknown mean $\hat{t}$ and finite standard deviation σ. The sample mean $\overline{T}$ can be approximated with a certain degree of confidence by a process of sampling. The standard error of the mean (SEM) will be used to get a confidence interval in which we can ensure the true mean is located. The standard error of the mean (SEM) is defined as

$$\epsilon = \frac{\sigma}{\sqrt{n}} \tag{11.1}$$

As the standard deviation σ is unknown, following the assumption that the error follows a normal distribution, we can use the student distribution to get the standard error using the standard deviation of the sample s

$$\epsilon = Z_\alpha \frac{s}{\sqrt{n}} \tag{11.2}$$

With a significance level $\alpha = 0.05$ we get from the t-student distribution the value $Z = 1.96$, and we can obtain the confidence interval $\overline{T} \pm \epsilon$ where we can ensure the true mean is located with a probability of 95%. By setting the relative error $\delta = \epsilon/\overline{T}$ to be lower than 1%, we obtain the limit error ϵ_0 to be $\epsilon_0 = 0.01\overline{T}$. Then, if we stop the simulation process when the standard error of the mean is below ϵ_0

$$\epsilon = Z_\alpha \frac{s}{\sqrt{n}} < 0.01\,\overline{T} \tag{11.3}$$

We can ensure that (a) with probability 0.95 the true mean $\hat{t}$ is located in the interval $\overline{T} \pm \epsilon$, and (b) the relative error of the mean δ is lower than 1%.

The process of simulation will run for at least a minimum of 30 iterations. Then it will continue until the relative error is below 1%, or the simulation time exceeds 30 minutes. All experiments were run in Mare Nostrum 4, using Intel MPI and the Intel `icc` compiler with the following modules:

- intel/2017.4
- tampi/1.0
- ompss-2/2019.06
- fftw/3.3.6
- impi/2017.4
- extrae/3.7.0

Use error bars to denote CI at 95% not std?

11.1 Performance model

Consider the real time of the simulation $\hat{t}(c)$ to be a function of a specific configuration c. There are a lot of parameters that may be changed and have some influence in the time per iteration, but we will focus only on the following ones.

- N_p: Number of total particles.
- N_g: Number of total grid points.
- N_c: Number of plasma chunks.
- P: Number of total MPI processes.
- C: Number of total cores.
- A: Whether TAMPI ($A = 1$) or MPI ($A = 0$) is being used.

A configuration is then completely specified as the tuple $c = (N_p, N_g, N_c, P, C, A)$. The space of states of configurations possible is bigger than the available time for experimentation, so we must choose a partial group which can reveal interesting information of the effect in the iteration time.

11.1.1 Number of particles

The number of particles N_p is one of the main parameters that affect the running time of each iteration as it can be observed from the simulation process that at least a complexity in $O(N_p)$ is expected—we need to cycle through each particle at every iteration. To get an accurate relation, an experiment is run sweeping from 2×10^6 to 4×10^7 particles, with 32 cores and only one process. The number of grid points is kept low at 1024^2 in order to avoid interference from the solver. We see in the figure 11.1 how the time scales linearly with the number of particles, and the residuals of the linear regression. With a determination coefficient of $R^2 = 0.99981$, we can estimate a time per particle of 51.4 µs with 32 cores.

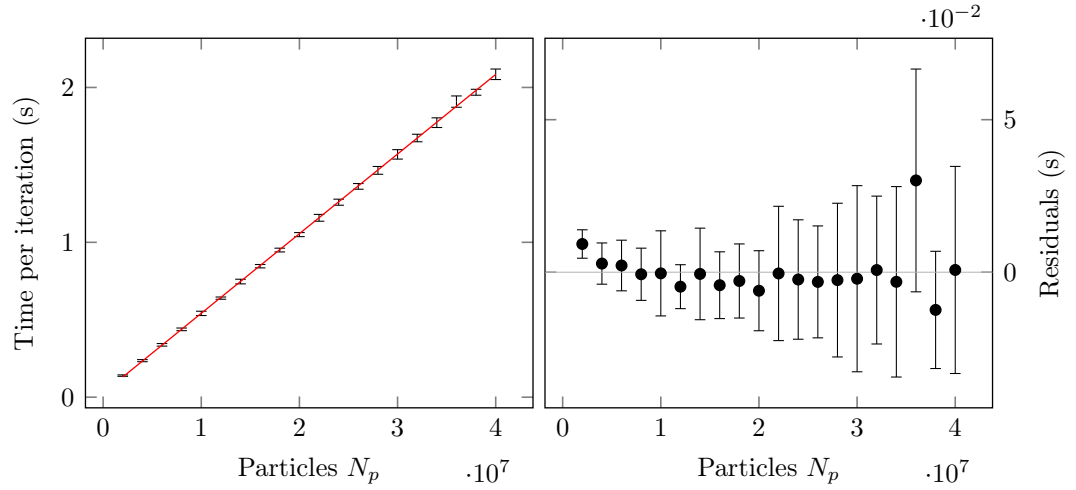


Figure 11.1: The number of particles N_p is increased and the time per iteration t is measured. A linear regression fit is shown, with the residuals at the left. Using one process and 32 CPUs MPI communications are not needed. Configuration used ($N_p = 2 \times 10^6$ to 4×10^7, $N_g = 1024^2$, $N_c = 128$, $P = 1$, $C = 32$, $A = 1$)

11.1.2 Number of grid points

The MFT solver uses the FFTW library to perform the FFT and solve the field equation in each iteration, with an expected worst time complexity in $O(N_g \log N_g)$. An experiment with varying number of grid points from 2048^2 to 8192^2 is designed to observe the grow in time. In the figure 11.2 it can be seen how the time grows with

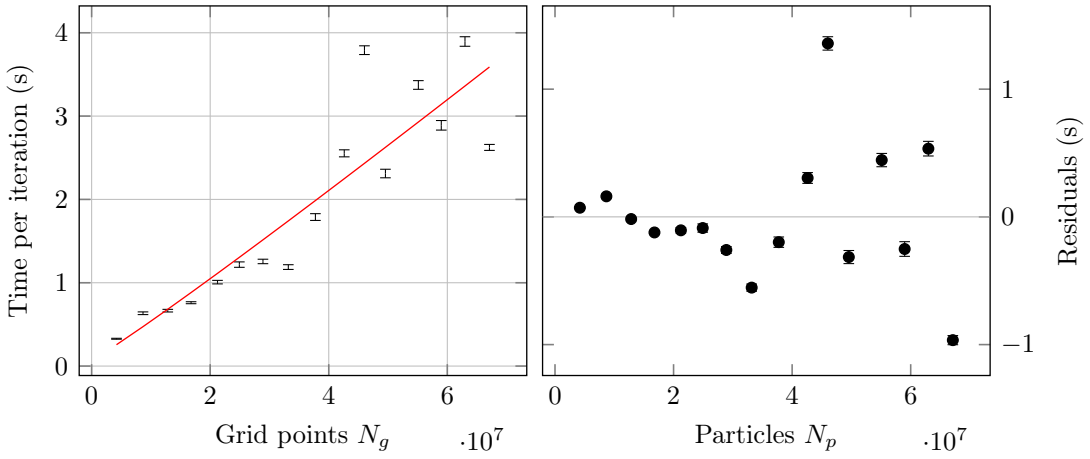


Figure 11.2: The effect of the variables N_p and N_g to the time per iteration. Using one process and 32 CPUs (MPI communications are not needed).

the number of grid points, but the variance is much bigger than with the number of particles. Notice that the dispersion is not due to random variations in the execution time, as the standard deviation for each point is lower than its dispersion from the common distribution. The FFTW uses an algorithm which benefits from sizes that can be decomposed into the product of small multiples ($2^a \cdot 3^b \cdot 5^c \cdot 7^d \ldots$). However the number of points in the X axis must be divisible by the number of plasma chunks, and some of the sizes tested had large primes in their decomposition.

Regenerate figure with appropriate sizes and check the dispersion decreases.

11.2 Solver multithreading scalability

The simulator is designed to scale with the number of particles when the number of cores or MPI processes are incremented—each chunk can be computed in parallel both in the X and Y axis. But when the number of grid points is incremented, the FFT solver must scale both in the number of CPUs and processes.

The space in Y is divided into equally sized blocks, which are assigned into MPI processes, following the parallelization design of the FFTW. Additionally the 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. Unfortu-

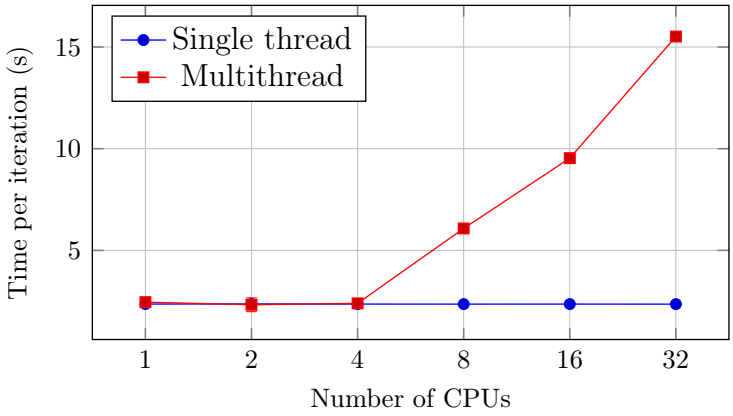


Figure 11.3: The number of CPUs is increased with only one process: the solver cannot scale and the time per iteration increases. Configuration used: $N_p = 5 \times 10^5$, $N_g = 8192 \times 8192$.

nately, the FFTW library doesn't show a good speedup, in fact worsens the time per iteration when adding more threads with the configurations tested. In the figure 11.3 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 the number of particles than grid points.

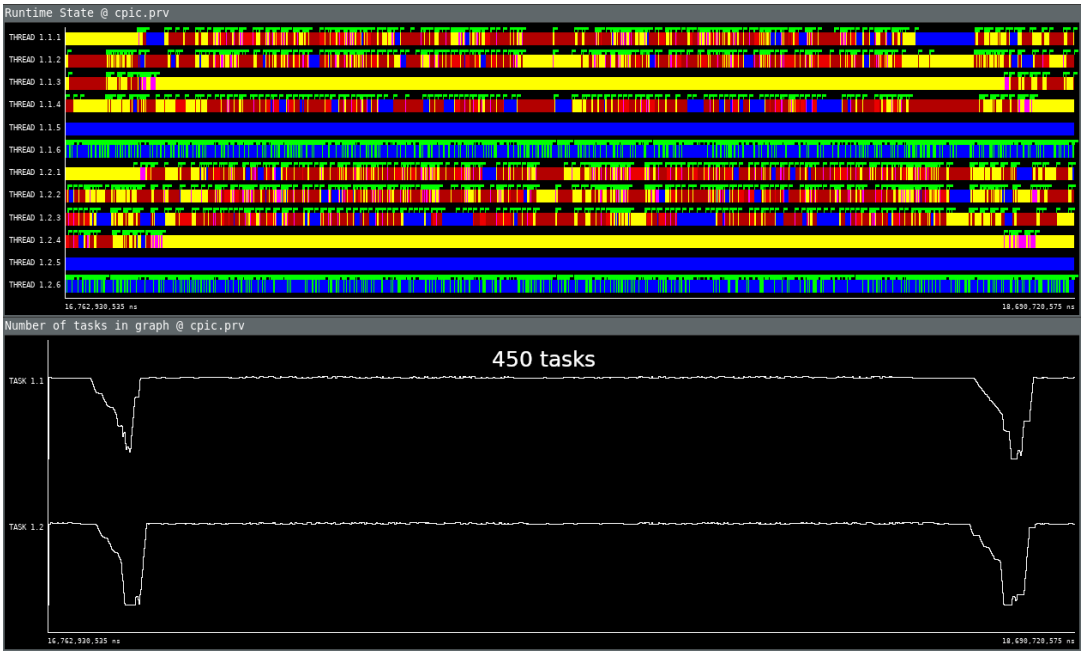


Figure 11.4: Tasks created inside the FFTW when using OmpSs-2: Up to 450 tasks are created in rapid succession, with only 4 CPUs and 2 processes.

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 being created. It shown that the overhead added by the large amount of created and destructed quick tasks outweight 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.

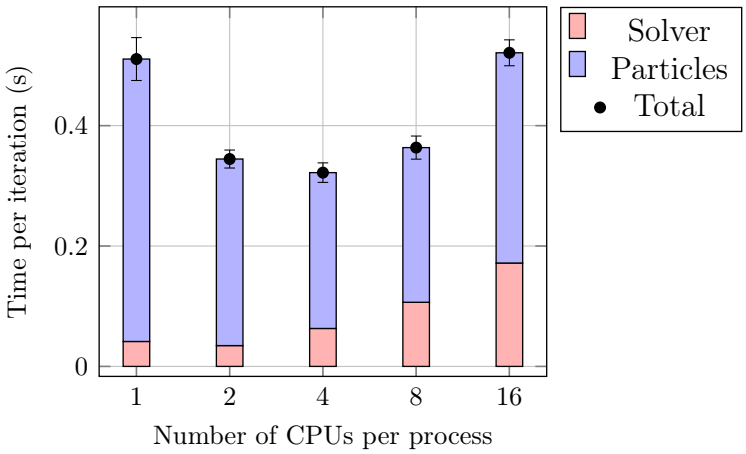


Figure 11.5: The number of CPUs per process is incremented while reducing the number of processes (the total number of CPUs is set to 32 and is kept constant). The time per iteration is measured, which leads to a characteristic U shape.

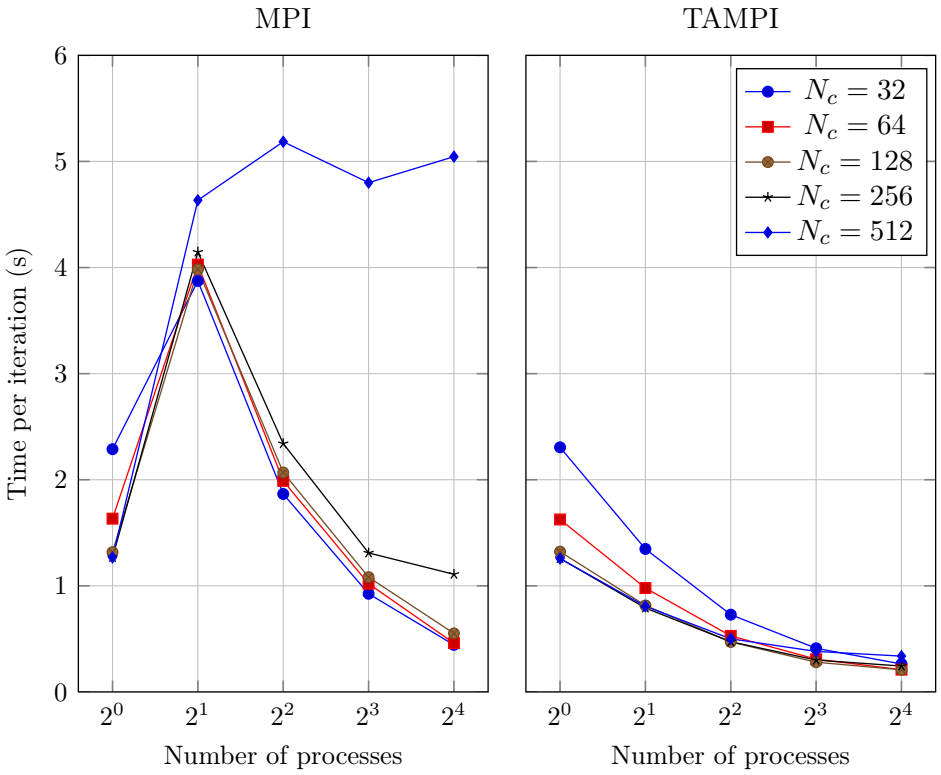


Figure 11.6: Comparison of MPI and TAMPI

11.3 TAMPI

The two modes of communication are compared with different configurations, in order to evaluate the effect in the overall performance of the simulation. The chunk size N_c determines the number of messages sent per process and is tested from 32 to 512. In the figure 11.6 it can be seen how the time is drastically reduced when TAMPI is enabled. The performance difference is lower when the number of processes increases, but is very significant with few processes. Notice the control case with only one process were no MPI nor TAMPI communication is needed (shared memory is used to exchange information between tasks).

It is also noted with MPI a saturation point with 512 chunks per process, where the time does not improve with more processes. Different versions of OpenMPI (3.1.1, 4.0.0 and 4.0.1) were also tested, and there was a extreme delay of more than one order of magnitude with respect to the mean time per iteration with low probability of occurrence, and the causes are yet unknown. The same problem was never observed with Intel MPI, but further investigation is needed to isolate the issue and conclude that is due to OpenMPI.

11.4 Scalability

In order to evaluate the simulator in terms of scalability the two main metrics are initially measured:

1. **Strong scalability**: The same configuration of problem is repeated with increasing number of computing elements.

2. **Weak scalability**: The number of computing elements is increased, while the amount of work assigned to each one is kept constant by changing the problem configuration.

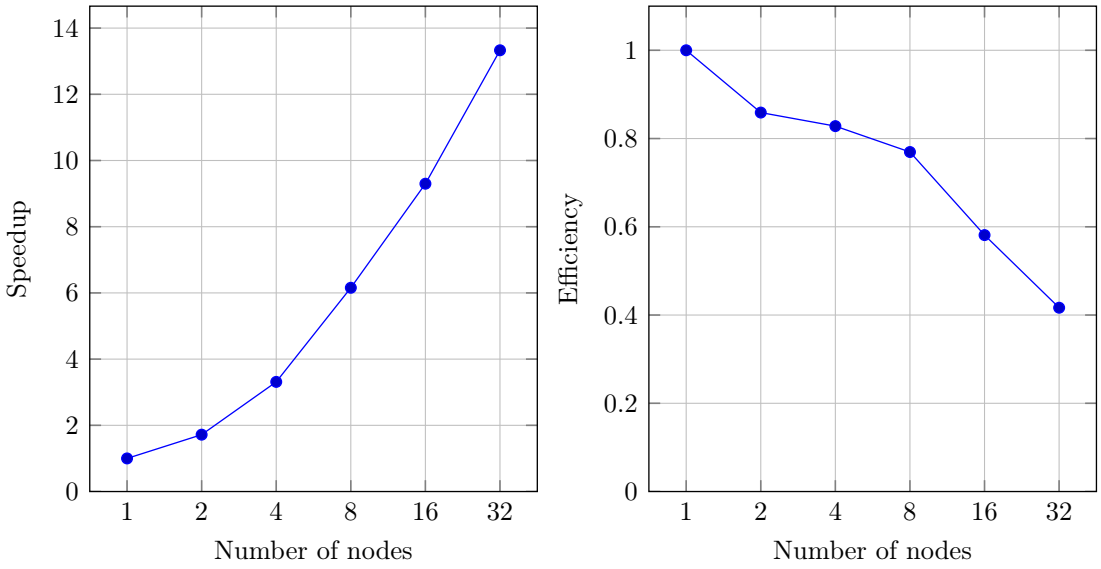


Figure 11.7: Strong scaling with configuration: $N_p = 1 \times 10^8$, $N_g = 2048^2$, $N_c = 128$, one process per node, using each 48 cores.

For the strong scalability, a fixed configuration of $N_p = 1 \times 10^8$ particles, $N_g = 2048^2$ grid points and $N_c = 128$ chunks is used to run the simulator with increasing number of computing nodes. Each node runs with 48 cores—all the available CPUs of the machine. In the figure 11.7 the speedup and the efficiency are shown. The rapid decay of efficiency is to be expected, as the solver cannot exploit the full 48 cores and only one is used when solving the FFT. We can obtain more information of the scalability of the simulator without the solver by disabling it—the physical result of the simulation will be non-sense, but the same stages of the simulator will be executed as if the solver was enabled. We see in the figure 11.8 how the efficiency now improves substantially, and indicates that the solver is acting as a bottle neck which leads to a significant reduction in scalability.

In order to analyze the weak scalability, a configuration is prepared to remain with constant work per computing elements (we will use the number of nodes, as each one will run at full capacity, using the 48 CPUs). The configuration chosen has 1×10^7 particles per CPU or 4.8×10^8 per node. The number of chunks is set to 128 and the number of grid points to 2048^2. Similarly as for the strong scalability, the number of nodes is tested from 1 to 32, in powers of 2. In the figure 11.9 the simulator shows a steady efficiency, which slowly decreases after the 8 nodes.

Test weak scaling with constant N_g per CPU.

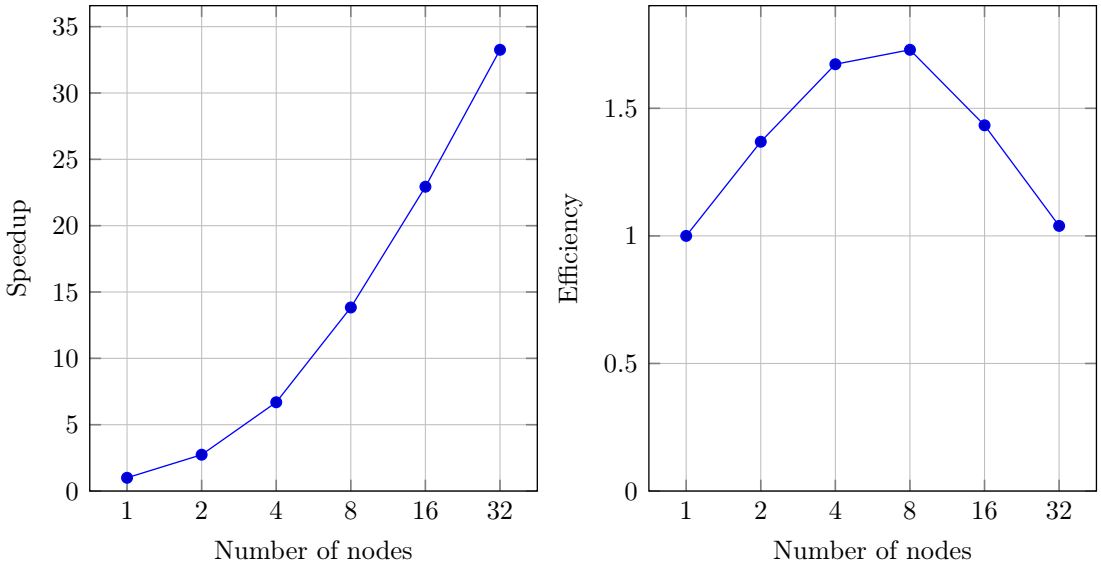


Figure 11.8: Strong scaling with the solver disabled (the physical simulation is incorrect without the solver, but the other stages of the simulation are properly executed as if they were genuine). Using the same configuration: $N_p = 1 \times 10^8$, $N_g = 2048^2$, $N_c = 128$, one process per node, using each 48 cores.

11.5 Extended scalability

In order to get more information when other parameters vary, more experiments were designed to show the effect in the efficiency. The solver is one of the main factors that adversely affect the iteration time, which can be mitigated by varying the ratio of CPUs per process, with the drawback of increasing the overhead in the other phases of the computation that benefit from shared memory communication, as identified in the figure 11.5.

Complete this

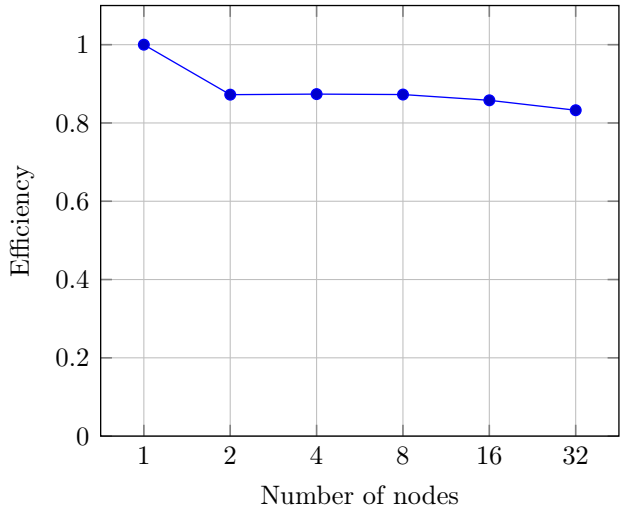
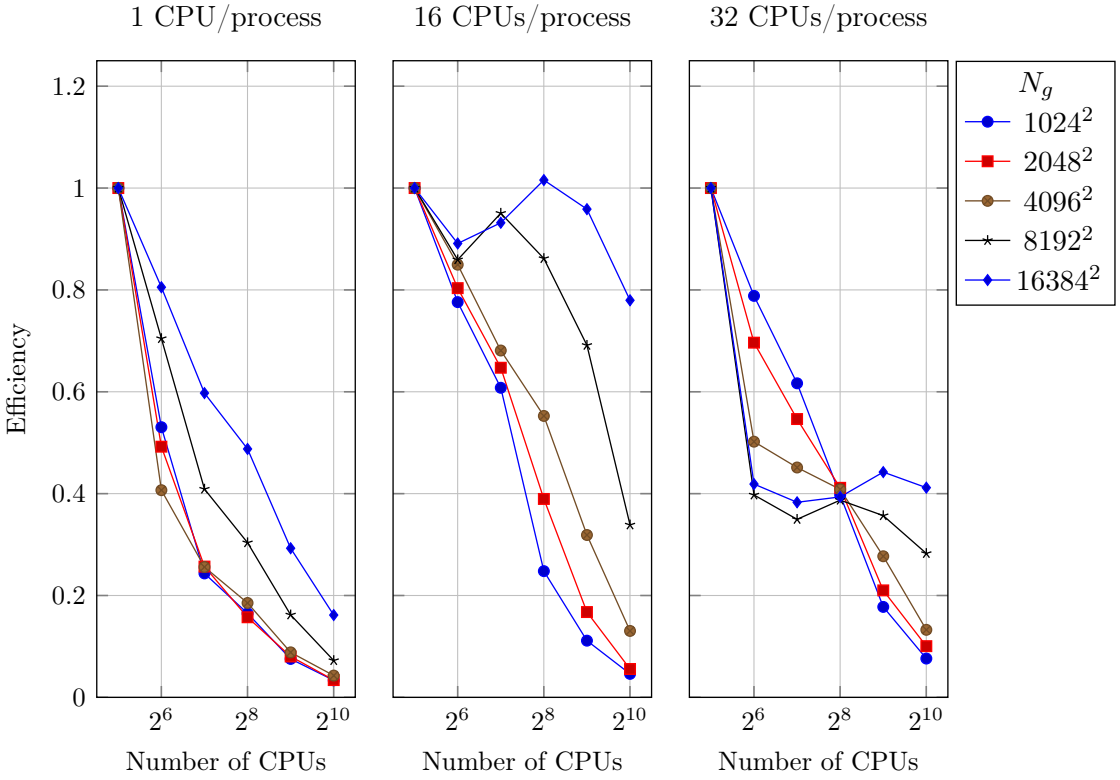


Figure 11.9: Weak scaling with 1×10^7 particles per CPU.



Chapter 12

Discussion

12.1 Conclusions

The presented simulator faces some challenges that are common in other real case scenarios, which were the main aim of this work. The parallelization of a complex simulation code cannot be fully optimal, if the programmer doesn't understand the whole process. For example, other solvers are available to find a solution to the field equations, but the spectral methods offer a very good theoretical complexity, which leads to the decision of how the space must be broken in blocks (split only one dimension), then the rest of the simulator design follows.

If the design were already chosen, the decomposition in tasks could only achieve a local optimum, depending on the methods used. Even in that case, the ease of task annotation is a low hanging fruit that can be tested without a big redesign.

Additionally, the TAMPI library was tested and compared with MPI, yielding better results with no big changes in the communication design.

Complete this

12.2 Future work

The main problem to be solved in the simulator is to address the scalability issues presented by the FFT, as the mitigations tested don't provide a good solution. One possibility is the interoperability of the OmpSs-2 runtime, nanos6, with external MPI processes with an additional mechanism of synchronization. In this way, the simulator can be fully parallelized, even at the core level. A step by step scheme for a configuration with C CPUs available per node and N nodes, is outlined as follows:

1. Begin the simulation as usual creating $P = N$ master processes, each with at least $N_c \geq 2C$ plasma chunks, to exploit the local parallelism of the C CPUs.

2. Place the fields ρ, ϕ and $\boldsymbol{E}$ in a shared memory region, accessible by other child processes.

3. Create K MPI child processes in each master process, with access to the shared memory and let them wait on a condition variable or the reception of a MPI

message. Ensure the number of points N_g in the vertical dimension is divisible by KP.

4. Continue the simulation until it reaches the solver stage.

5. Ensure all tasks are finished, and wake all the child processes and then wait for them to finish.

6. In each child process execute the distributed FFTW with KP processes, and use the shared memory to access the fields.

7. Once the FFT finishes, signal the master and put each child process to sleep again, waiting for a signal.

8. In the master process, the ϕ field is now ready in the shared memory region. If the simulation is not finished, go to step 4.

The key concept is that we are moving temporally the threads of the OmpSs-2 runtime away from the CPUs to let the MPI processes of the FFTW take control of the full parallelism using all the available CPUs. No change is needed in the FFTW library, and this method may benefit other programs with similar issues.

On the other hand, the physical results must be validated with a direct comparison with other simulators, as is very easy simulate non-realistic behavior without noticing. The different validation techniques provide some ground that the simulation follows the expected behavior, but don't guarantee any correctness.

Additionally, there are a large list of improvements that were planned, but may be improves in a future work:

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

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.