Porting PIC Code from Pthreads to MPI

Snorre Boasson

October 10, 2003

Contents

Chapter 1

The Particle in Cell method

In physics, there are many situations in which we have many objects who are all exerting forces on each other. One example is the universe, where we have vast amounts of objects with mass. According to the law of gravity, all these objects will pull on one another. The force two arbitrary objects exert on each other is inversely proportionate to the distance between them. Thus, if we regard this as a system with N objects, we would have to compute $N * (N - 1)$ relations, which amounts to a runtime of $O(N^2)$.

Another example is plasmas, in which charged particles will exert electrostatic forces on one another.

Common for both systems, is that we will often find ourselves interested in modelling them with many millions, or even billions of objects, or particles. In which respect the $O(N^2)$ runtime of the direct approach a real show stopper.

The *Particle-in-Cell* (PIC) method supplies us with a method of simulating such systems. Instead of solving for each particle-to-particle relation, the effect of each particle is interpolated onto a grid. In a plasma simulation, this means interpolating the the charge of a particle onto the grid. Thus, the grid becomes a discretised representation of the charge (or mass) distribution in our simulation space.

The macroscopic effects can then be solved on the the grid. This generally means solving a 2nd order partial differential equation (PDE). In the plasma case, this enables us to determine the strength and direction of the electrical field at the grid points. The field is then interpolated back onto the particle, which is then moved according to the forces exerted upon it.

The PIC simulation is consists of two parts; an initialisation and a simulation loop. The reason an initialisation is needed is the fact that as the particles are in place, the electrical field and forces working on them is there instantly. Thus, we need to run through the sequence once, dropping the particle moving, to get a realistic starting point. Then, we can start the simulation time, and let the simulation commence.

1.1 Particle to grid transition

The particle-to-grid transition is done by calculating the charge contributed by each particle in its four adjacent grid points. The charge contributed is calculated as

$$\rho_{gp} = \rho_{gp} + \Delta x * \Delta y * \rho_{particle} \tag{1.1}$$

where Δx and Δy are the distance from the particle to the grid point in x and y

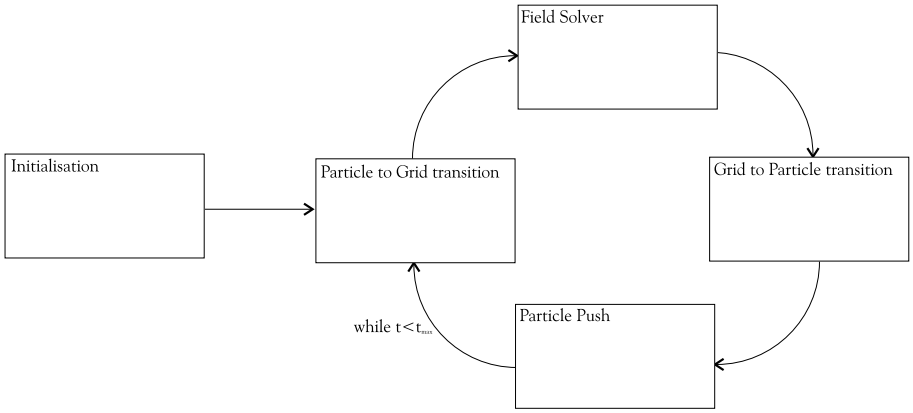

Figure 1.1: This shows the flow of a PIC simulation. After a initiation, the four steps are repeated until the end of the simulation time is reached.

respectively, ρ_{gp} is the charge at the grid point in question, and $\rho_{particle}$ is the charge of the particle.

1.2 GRID COMPUTATIONS

1.2.1 *Calculating the potential*

From `PART_RHO` we now have an approximation of the charge distribution in the simulation space, in form of the $\rho_{x,y}$ array. From this, we now need to calculate the electric potential Φ on each grid point. Mathematically, this is given by:

$$\nabla^2 \Phi = -\frac{\rho_{x,y}}{\epsilon_0} \tag{1.2}$$

which can also be written in the form

$$\frac{\delta^2 \Phi}{\delta x^2} + \frac{\delta^2 \Phi}{\delta x^2} = -\frac{\rho_{x,y}}{\epsilon_0} \tag{1.3}$$

which is a Poisson's equation. A Poisson's equation is a linear partial differential equation of the second order. This can be solved directly, using the Fourier transform, or we can use an iterative numerical solver.

1.2.2 *Determining the electrical field*

Having found the potential $\Phi_{x,y}$ we now need to calculate the electrical field. As an electrical field has both strength and direction, these are vector values. They are calculated using one-dimensional (1D) difference equations:

$$\vec{E_{x(i,j)}} = \frac{\phi_{i,j-1} - \phi_{i,j+1}}{2 * h_x} \tag{1.4}$$

$$\vec{E_{y(i,j)}} = \frac{\phi_{i-1,j} - \phi_{i+1,j}}{2 * h_y} \tag{1.5}$$

where h_x, h_y are the grid spacings in x and y respectively.

1.3 GRID TO PARTICLE TRANSITION

Now the field is interpolated back to the individual particle positions. The interpolation formula is the same that was used before:

$$E_{particle} = \sum_{for\ all\ corners} E_{gp} * \Delta x * \Delta y \tag{1.6}$$

All the four corners of the cell contribute to the field at the particle's location. Δx and Δy is the distance to each grid point in x and y, E_{gp} is the field in x and y at the grid point.

1.4 PARTICLE COMPUTATIONS

Knowing the forces affecting the particle, it can now be moved with the use of Newtonian physics. Different models can be used. The particular method used in Elster's code was leap frog (see [3] for details).

Chapter 2

The Parallelisation of PIC codes

The parallelisation of PIC codes involves the parallelisation of two sets of computations - the particle computations and the grid computations. These two sets has different parallelisation traits. Strictly speaking, the particle computations can be trivially parallelised by dividing particles equally across the processors. However, for the grid computations, the locality of the grid data is important. Thus, one will have to take particle locality into account, or choose to communicate grid data before starting the grid solving phase.

Generally, one can divide the possible decompositions into three classes, often called *Lagrangian*, *Eulerian* or *Adaptive Eulerian* decompositions [3].

Lagrangian decomposition means having a static division of the particles between the processors. No regard to particle location is made, and the field equations can be solved in an arbitrary manner.

Eulerian decomposition is when the grid is divided between the processors. Particles are handled by the processor in charge of the grid area in which they reside. This means that as the simulation progresses, particles will often have to migrate from one processor to another.

Adaptive Eulerian. Because pure Eulerian decomposition lead to quite a load balance problem, a solution is to repartition the grid. The aim of the repartitioning is to try and make the particle count as even as possible between the partitions.

2.1 Lagrangian decomposition

As mentioned, Lagrangian decomposition means having a fixed number of particles assigned to each processor. Each process has a local copy of the complete grid. After particle contribution to the grid is calculated, all the local grids are summed up to produce the global result. This is then spread to all processors, and the field is solved. The exact details of how the global results are spread, and how the field is solved will be dealt with a bit later. For now we assume all processors have a complete copy of the resulting field at this point. Next, the processors can then update their particles, and the next time step ensues. Adapting a serial PIC code to a Lagrangian scheme is thus quite simple:

- Divide the N_p particles out on the p processors. Each processor then has my_N_p particles.

- After each call to the particles-to-grid function, sum up the local results to obtain the global grid result.

The Lagrangian decomposition leaves us with quite free hands as to how we choose to solve the grid equations. The simplest strategy is of course to simply let each processor solve it on its own. This is wasteful, however, but if parallel solvers calls for extensive communication, it could still be a viable option. In this case, one would sum up the global result of the grid, and broadcast it to all processors. This could be done with one call to the `MPI_Allreduce()` function. However, we need to closely watch the Particle-vs-Grid computation ratio, or else it could spell doom on the efficiency of our simulation.

Another problem is the fact that we will wind up with many partial results scattered across the processors, and they will need to be summed up. The complexity of this sum operation increases with the number of processors, making Lagrangian schemes will scale rather poorly, as pointed out in [3]. Thus, to obtain a more scalable code, other decomposition strategies will have to be explored.

2.2 EULERIAN DECOMPOSITION

Using an Eulerian decomposition, we divide the simulation space between the processors. Effectively, it amounts to dividing the grid cells between the processors, and then assigning the particles to the processor "owning" the cell it resides in.

The motivation behind this strategy is to eliminate the need for summing up partial, local results on the grid. It is parallel overhead, after all. In addition, it ensures a natural data location, that can be exploited by a parallel field/grid solver.

However, the division of the grid now calls for a mechanism to allow particle migration between processors. After all, the particles does not adhere to the division, they must be free to roam about as they will.

Simply put, after the particle push phase (the updating of particle positions, that is), a processor would need to round up all particles that have left it's domain, and send them off to the processor in charge of the domain they have entered. This would be accomplished by sending two messages, one telling the recipient how many particles. The next message would then contain the particle data. This strategy is also used in [4]. The reason for the two-message approach is that MPI does not support the sending of messages that the recipient does not know the length of.

Another question that needs to be addressed is how to deal with particles in border cells. The grid division is done on the grid points. Thus, as shown in figure 2.1 the cells on the border lines have grid points belonging to two or four processors. Thus, particles residing in these cells will contribute to the charge in grid points on several processors, which must be handled in some way. At first glance, two options seems viable. Either having a copy of the particle on each of the affected processors, or exchanging local grid results along the border lines.

Replicating particles is one possible venue. The idea is that as a particle enters a border cell, it's information is sent to the neighbouring processor(s). The simulation could then continue. If or when the particle leaves the border area, all processors but the one whose domain it enters simply drops it. By doing this, one could hope avoid more communication than the particle migration that would have to be undertaken in sooner or later in any event. But as it turns out, it will invariably lead to more communication

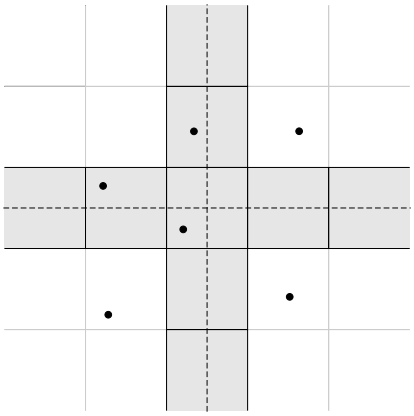

Figure 2.1: This shows how cells becomes divided between processors. The gray cells are border cells, the dotted lines shows the processor domain division.

than simply exchanging the border values. This becomes obvious by looking at how the processes would update a such shared particle at time steps. In order to determine it's new location, the field values of all the grid point adjacent to the cell would still be needed. Which leads us back to where we started, without improving anything in the process. In addition, other this strategy leads to other problems as well. In the border cells there would be four copies of each particle - how would one determine which one to use in e.g. visualisation or other output? This would call for extra memory use for each particle, to keep track of "master" particles and "ghost" particles. To sum up - it was an interesting idea, but it did not stand up to further scrutiny.

Exchange of border values is then the strategy we are left with. This is rather straight forward - at each time step, the border values are sent to the neighbours adjacent to that border. Some sort of round-robin could be arranged in order to avoid having to send and receive at the same time, or one could use the "immediate" versions of send and receive to do both at the same time. Which one would yield the best performance would have to be tested for, as this depends heavily on the specifics of the hardware on which the code is running. The particles would only be migrated once they enter a grid area completely owned by another processor. Depending on the solver decided upon, one could even find the values already exchanged as a part of the solving process. This possibility will be further addressed in the discussion of different field solvers.

To incorporate an Eulerian decomposition into a PIC code, some additions are needed. First, we need some book-keeping to keep track of particles going in and out of the domain. In addition, there will have to be some extra code at the start determining the division of the grid. The point wise list contains the necessary steps. Some points may be interchanged here and there, so the ordering is not strict, but it represents a natural ordering for actions needed. The variable names suggested are of course optional, but we will use them later in this thesis.

- Read in global parameters, `Nx`,`Ny`, `Np` and the constants needed (see table 3.2).
- Factorise the number of processors, `num_procs` into two factors, as equal as possible. This step can be eliminated by having the user specify number of processors

in x and y direction.

- For each processor, determine how large it's part of the grid will be. As this is *pure* Eulerian, we do not consider load balancing at this point. Thus, we aim to make the division as equal as possible. The local size is recorded as `my_Nx` and `my_Ny`.
- To know where in the global grid the processor's sub grid belongs, we store the lower left corner's global coordinates in `llc_x` and `llc_y`.
- Each process needs to know the rank of its eight neighbours (four on each side, plus four corners). To this end, an array, `int *neighbours`, 8 ints long, is utilised. In this array we store the ranks of the neighbours, as counted clockwise from the upper left corner.
- The array used to store the local sub grid, is increased by 2 in each direction. This is to keep track of what's happening in the border cells.
- The particles are distributed across processors according to grid. The number of particles assigned to each process is recorded as `my_Np`
- The original `PART_RHO()` is called, with `my_Np` particles. The grid sizes in X and Y is (`my_Nx`+2) and (`my_Ny`+2), respectively.
- The border lines are exchanged, making each process possess the global results for their ghost points.
- After each update of particle positions, the particles leaving that have left their original domain must be migrated.

As we see, this was a lot more complicated than the Lagrangian decomposition.

One grave issue with a pure Eulerian approach is load balancing. Unless the simulation parameters results in a relatively even distribution of the particles, then some processors will wind up with more particles to handle than others. Theoretically, one processor could end up handling all the particles, which would effectively return the simulation to a serial one. This would not only affect runtime, but makes it hard to predict how much memory is needed by each processor for a simulation. In order to avoid memory problems, one would therefore be forced to reduce the number of particles in the simulation, or the simulation code would need some backup solution in case one process cannot handle all the particles within its domain. These are grave problems, as they undermine the whole purpose of parallelising the simulation.

2.3 ADAPTIVE EULERIAN DECOMPOSITION

To combat the problems arising from a pure Eulerian decomposition, one can repartition the grid at intervals. The purpose of this repartition is, obviously, to try to make the number of particles within each partition as even as possible. A number of algorithms could be employed to this end.

As one has determined/constructed some load balancing functions to use, they should be implementable under the scheme drafted for Eulerian decomposition. None of the functions there operate under the assumption of any equal division of the grid. Thus, a load balancing function could be implemented into `eulersplit()`, and this

could then be called at certain intervals. After a new decomposition has been made, `migrate()` should (or quite likely must) be called, to migrate particles that are no longer within a process' jurisdiction.

2.4 OTHER COMBINATIONS

As has been seen, Lagrangian decomposition is automatically load balanced, but it does not scale well. Eulerian decomposition has the promise of scaling much better, but it creates a load balancing problem.

Strategies to combine them could therefore be of interest. The main Achilles heel of the Lagrangian decomposition, is that each process will sit on a part of the $\rho_{x,y}$ grid, and all these local results will have to be summed up in some manner.

To alleviate this situation, one could imagine making a division of the grid, but not divide the particles according to it. After the ρ matrix has been computed, the local results for each grid part would then be sent to the owner of that part. Whilst not doing completely away with the problem of having many local results needing to be summed up, it does away with the all-to-all communication.

In hierarchical clusters a combination could also be a smart solution. Internally on SMP nodes, one could run Lagrangian, as these nodes usually boast two or four processors. On the distributed memory level, one would resort to an Eulerian decomposition. This strategy could also be used on the shared memory level. This would mean using a Eulerian decomposition on one level, then then assigning groups of 2-8 processors to each division. These groups then use a Lagrangian decomposition between them.

Chapter 3

Anne C. Elster's PIC code

In this chapter, the different bodies of code that have been implemented will be described. First, each part of Anne C. Elster's original PIC code will discussed. This is necessary, as the new code will use different parts of Elster's code as building blocks, as suggested in [2]. These parts are presented as a body of pseudo code algorithms, which will then be used to discuss what additions and changes has been made, as well as help the understanding of how the different parts come together to form the complete PIC loop.

Also note that the information in the first section of this chapter is presented in with an intended redundancy. Tables, pseudo code algorithms and textual descriptions will all contain some information present in the other parts of the presentation. This intention behind this is to try to make the subject matter as easily available to the reader as possible, without going into full-fledged "parrot mode".

The PIC code by Elster simulates a two-dimensional (2D) plasma with periodic borders. It has been partially parallelised using POSIX threads, to run on a shared memory architecture. The parts that were parallelised was the functions dealing with particle computations; i.e. `PART_RHO()`, `PUSH_V()` and `PUSH_LOC()` These parts where, however, commented out, and were totally removed before the MPI parallelisation begun. The first MPI parallelisation was, however, a remodelling of that original parallel version. All the pseudo code presented here is modelled after the serial version, though.

Table 3.1: Overview of the arrays used for table storage.

Array	Size	Description
`*Phi`	N_g	Potential at each grid node
`*Rho`	N_g	Cumulative charge at each grid node
`*Ex`	N_g	Electric field strength in direction X
`*Ey`	N_g	Electric field strength in direction Y
`*Part_x`	N_p	X values for particle locations
`*Part_y`	N_p	Y values for particle locations
`*Vx`	N_p	Velocity vector in X direction
`*Vy`	N_p	Velocity vector in Y direction

Table 3.2: **Constants used in the simulation** (Note: Dual constants, where there are one for each dimension, is put on the same row, but they still count for two)

Size	Name	Description
`int`	`Nx, Ny`	Number of grid points in X and Y
`int`	`Ng`	Total number of grid points (=`Nx*Ny`)
`int`	`Np`	Total number of particles
`int`	`Px, Py`	
`double`	`eps`	
`double`	`mass`	
`double`	`q`	Particle charge
`double`	`rho0`	
`double`	`rho_k`	
`double`	`drag`	Drag constant.
`double`	`Lx, Ly`	Total length of grid in X and Y
`double`	`hx, hy`	Length of grid cell in X and Y
`double`	`t`	Current simulation time
`double`	`del_t`	Size of time step
`double`	`t_max`	Simulation end time

3.1 ARRAYS AND CONSTANTS

All the simulation data is kept in 1D arrays of doubles, as shown in Table 3.1. For the 2D arrays, the conversion from a coordinate (i, j) to the array entry `n` is simply done as

```
n = (i*Nx+j)
```
(3.1)

From Table 3.1 we can also find an expression for the memory used to hold the particle and grid data. We have eight arrays; four of length N_p and four of length N_g. Memory usage is then

$$4(N_p + N_g) * 8bytes \tag{3.2}$$

which is a linear relation to the number of particles and number of grid points.

The simulation also uses several constants. These are shown in table 3.2. From the table we see that there are 6 `int`'s and 13 `double`'s. Together, this takes 128 bytes of storage, which in today's computers with gigabytes of memory is rather negligible. In addition there will be some variables used for program control; loop counters etc. These will not be the subject of any separate discussion, but will be mentioned where appropriate.

3.2 FUNCTIONS

Some of the functions used in the simulation code has already been mentioned. Table 3.3 shows the simulation functions used, and what arrays and constants they use.

For each C function, the parameters passed are divided into 3 categories; *inputs*, *constants* and *outputs*. This division is meant to help understand the information flow, as these distinctions are not explicitly shown in C code. Also, note that *some constants* are listed *as inputs*, because *they are important in the parallelisation of the code.*

In the pseudo code algorithms, inputs are listed after the function names. Outputs are specified using "return" statements at the end of the algorithm.

Table 3.3: **Simulation functions** Note: Some functions have certain arrays listed as both in and out parameters — this means that they need the old value, but will update them.

Name	Constants	Input(s)	Output(s)
`PART_RHO()`	`Np, Nx, Ny, rho_k, hx, hy`	`*Part_x, *Part_y`	`*Rho`
`FFT_SOLVE()`	`eps, Lx, Ly, Nx, Ny`	`*Rho`	`*Phi`
`PERIODIC_FIELD_GRID()`	`Nx, Ny, hx, hy`	`*Phi *Ex, *Ey`	
`PUSH_V()`	`Np, Nx, Ny, Hx, Hy, drag, q, mass, del_t`	`*Ex, *Ey, *Part_x, *Part_y, *Vx, *Vy`	`*Vx, *Vy`
`PUSH_LOC()`	`Np, Lx, Ly, del_t, t`	`*Vx, *Vy, *Part_x, *Part_y`	`*Part_x, *Part_y`

Constants are the values needed by the functions in their calculations, that are not affected by the simulation itself. While constants of importance to the parallelisation is listed as inputs, they are still constants in all senses of the word in the serial version.

These pseudo code algorithms will then serve as a bridge from the mathematics discussed in Chapter 3, and to the implemented C programs found in the appendix.

Note that this pseudo code will follow the structure of the C code closely, even though it does not follow C syntax. For brevity, all calculation steps have been explained by text instead. For mathematical details, reference to the equations in Chapter 3 is given. For C code, reference in line numbers are given.

3.2.1 *PART_RHO*

As has been mentioned previously, the `PART_RHO` function is in charge of discretising the (continuous) cloud of particles into the discrete grid.

Also note that the algorithm is listed with Np as input. This distinction is made as we will pass the variable `my_Np` instead of `Np` to this function when running the code in parallel.

Algorithm 1 `PART_RHO(Np, *Part_x, *Part_y)`

Require: `Nx, Ny, rho_k, hx, hy`

1: **for all** particles from 0 to `Np-1` **do**
2: determine lower left corner of cell `i,j`
3: determine distance to left and lower wall `a,b`
4: calculate contribution to all four corners, wrap around for borders, insert into `*Rho`
5: **end for**
6: **Return** `*Rho`

3.2.2 *FFT_SOLVE*

The FFT solver function's sole purpose is to solve the Poisson's equation. The inner workings of the FFT transform is beyond the scope of this text. But we represent it as a code block here, so it can be used as reference. The FFT solver should probably be substituted by a parallel CG (Conjugate Gradient) method for parallel solving.

Algorithm 2 `FFT_SOLVE(*Rho)`

Require: `eps, Lx, Ly, Nx, Ny`

1: Fourier Transform
2: Solve
3: Inverse Fourier Transform
4: **Return** `*Phi`

3.2.3 *PERIODIC_FIELD_GRID*

This function uses a 1D difference equation to compute the field vectors in X and Y dimensions, respectively. The grid size constants `Nx` and `Ny` are listed as input as they will be substituted for parallel application.

Algorithm 3 `PERIODIC_FIELD_GRID(Nx, Ny, *Phi)`

Require: `hx, hy`

1: **for all** grid points (`Nx*Ny`) **do**
2: Calculate `Ex, Ey`
3: **end for**
4: **Return** *Ex, *Ey

3.2.4 *PUSH_V*

To update particle velocities, `PUSH_V` is called. It takes many of the simulation arrays as input. The velocity arrays are both input and output as the old values are used to calculate the new ones.

Algorithm 4 `PUSH_V(*Ex, *Ey, *Part_x, *Part_y, *Vx, *Vy)`

Require: `Np, Nx, Ny, hx, hy, drag, q, mass, del_t`

1: **for all** particles from 0 to `Np-1` **do**
2: Interpolate field to particle location
3: Calculate forces exerted on particle
4: Calculate new velocities
5: **end for**
6: **Return** `*Vx, *Vy`

3.2.5 *PUSH_LOC*

With the new velocities, `PUSH_LOC` can update the particle positions. As the new positions naturally are dependent on on the old ones, the location arrays serve as both input and output.

Algorithm 5 `PUSH_LOC(*Vx, *Vy, *Part_x, *Part_y)`

Require: `Np, Lx, Ly, del_t, t`

1: **for all** particles from 0 to `Np-1` **do**
2: Calculate new position
3: **end for**
4: **Return** `*Part_x, *Part_y`

3.3 PUTTING IT ALL TOGETHER: A SERIAL PIC CODE

Having the different simulation functions pinned down, we will use these to build a complete PIC code. In figure 3.1 the blank boxes in figure 1.1 from chapter 3 has been filled in. As we see, one part of the simulation has not been much mentioned, the initialisation phase. Looking at it, we see that it even contains some unmentioned functions, and some of the simulation functions as well. Which will all become clear in a moment.

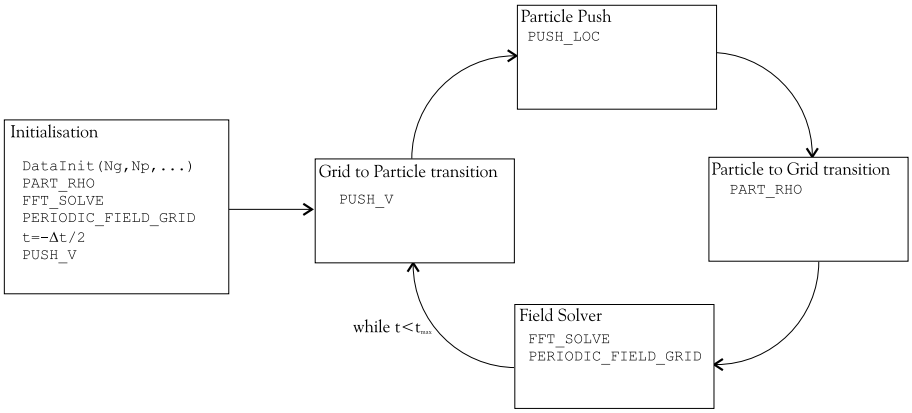

Figure 3.1: PIC code and its building blocks

3.3.1 *Data initialisation functions*

The PIC code by Elster utilised two data initialisation functions. Their job is quite simply that of reading in start values for constants from files, and zeroing out all arrays. [comment:more on this]

3.3.2 *Further initialisation*

Some of the simulation functions were also used in the initialisation phase as well. The reason for this is quite simple; as the simulation starts, there is already a field there due to the particles. Without the first run of simulation functions, the start conditions would be rather unnatural. To build a field, `PART_RHO`, `FFT_SOLVE` and `PERIODIC_FIELD_GRID` is called. Then, in accordance with the Leap Frog method, simulation time is pulled back half a time step, and the initial velocities are modified through `PUSH_V` to reflect their values at `T` minus $\Delta t/2$. (`T` being time zero, as NASA launch count down tradition dictates).

Algorithm 6 Serial PIC

Require: all

```
1: Initialise Data
2: *Rho = PART_RHO(Np, *Part_x, *Part_y)
3: *Phi = FFT_SOLVE(*Rho)
4: *Ex, *Ey = PERIODIC_FIELD_GRID(*Phi)
5: t = -Δt/2
6: *Vx, *Vy = PUSH_V(*Ex, *Ey)
7: for all t from t=0 to t=t_max do
8:   *Vx, *Vy = PUSH_V(*Ex, *Ey)
9:   *Part_x, *Part_y = PUSH_LOC(*Vx, *Vy)
10:  *Rho = PART_RHO(Np, *Part_x, *Part_y)
11:  *Phi = FFT_SOLVE(*Rho)
12:  *Ex, *Ey = PERIODIC_FIELD_GRID(*Phi)
13:  t = t+del_t
14: end for
```

3.4 PARALLEL PIC USING THREADS

Elster's code was originally multithreaded, allthough the multithreading parts was rem'ed out of the code as it was when this work on it started.

Parallel programming with message parsing makes the distributed memory of the underlying hardware much more explisit. On a cluster of workstations, the distributed nature of the memory is also much more explisit in a physical way, in that the "Supercomputer" is acutally no one computer, but separate PC's networked together.

With MPI, no information is shared from the get go apart from command line parameters given at startup. All other information will have to be explisitly acquired by each process. This can either be done by message passing, but for start parameters it can be organised by having all processes read them in from disk. This makes the look of a threaded parallel program, and a message passing one rather different.

The thread parts of the original code, however, clearly showed the how each thread was assigned a sub set of the particles in the particle computation steps. This gave good hint at how to start with building a Lagrangian MPI parallel version.

But where the threaded version could simply figure out where in the list of particles it should start it's work, and how many particles to process from that point, the MPI version would have to have each process posess separate sets of particles.

Thus the code, along with the more theoretical descriptions of parallel decomposition schemes would serve more as guidance, than as something that could be directly translated to MPI parallelism. It was therefore decided to strip the program down to a pure serial one, and build the parallel version from the building blocks of this version.

Chapter 4

Parallelisation with MPI

Having dissected the original PIC code, we will now begin putting together pieces in a manner that is suitable for message passing parallel execution.

Keeping in mind that the goal of this is not necessarily one PIC code that is all dressed up and ready to go. Rather, the mission is to explore different strategies, and their performance on a cluster, and when possible, compare it to the performance on a supercomputer.

Serving as starting point once again, we have the `PART_RHO` function. With its role at the transition from particle to grid, it is the natural point to implement parallelisation routines. The grid-to-particle transition does not give the same opportunity, as the grid computations are just left behind, and grid decomposition will not be an issue with regards to the field solver functions until `PART_RHO` is called again. So, the focus on this function has its reason.

The pseudo code "algorithms" are here meant to be coarse representations of whole programs. They can be viewed as as an pseudo code equivalent to the C `main()` function. The parameters listed as **Required** are the data the `main()` function will have to provide. The "return" statements will be used to show both what data is produced, as well as what timing results are investigated.

In order to see the impact of the parallelisation overhead on the PIC code, we need to know how long time is spent calculating, and how long is spent administrating the parallel execution. In order to determine this, we need to run the whole set of computations. For our purposes, it is not critical if there are some parts of coding "duct tape" to complete the loop. Although not desirable, such parts will be timed, and then subtracted from the total. The aim is to analyse the cost (in time) of the parallel-specific operations, and see how they compare to the time spent by the simulation calculations.

4.1 Lagrangian decomposition

The first parallelisation strategy tested out was a Lagrangian scheme. Recalling from ch. 2, a Lagrangian decomposition scheme involves replicating the grid on all participating processors. The particles are simply split evenly across the processors, with no regard to their locality. Each process then computes a charge distribution grid based on their local particles. All the local grids are then summed up using the MPI function `MPI_Allreduce()`.

4.1.1 A Lagrangian test of PART_RHO

This simple program's purpose was to give an idea as to how long the `MPI_Allreduce()` call would take compared to the run time of `PART_RHO`, for varying grid sizes.

The `*Rho` matrix is emptied before each iteration of the simulation loop. This might be a natural thing to incorporate in the function itself, but it was placed outside in the original code. In order keep the two as alike as possible, to better hi light the parallelisation, it has been kept so. For brevity it is left out of the pseudo code algorithms.

Algorithm 7 Lagrangian PART_RHO

Require: Np, Nx, Ny, *Part_x, *Part_y
1: my_Np = Np/num_procs
2: Local Rho result = PART_RHO(my_Np, *Part_x, *Part_y)
3: Global Rho result = MPI_Allreduce(Local Rho result)
4: **Return** Timing step 2, Timing step 3

The results are encouraging. As we see, for moderate grid sizes, the time used by the `MPI_Allreduce()` call are reasonably small. And as integrating these changes into a complete PIC code is very simple, we go ahead with it.

4.1.2 A Lagrangian PIC code

With the groundwork done on making a parallel stand alone of `PART_RHO`, it is very easy to make a Lagrangian version of our PIC code. Simply, as you have divided the particles evenly across the processors, just add a call to `MPI_Allreduce()` after each call to `PART_RHO` in the main simulation loop. Make sure all particle calls are given the number of local particles, and not the global total, and you're done. In pseudo code:

Algorithm 8 Lagrangian PIC

Require: all
1: Initialise Data
2: `my_Np = Np/(number of processors)`
3: `*Rho_local = PART_RHO(my_Np, *Part_x, *Part_y)`
4: `*Rho_global = MPI_Allreduce(*Rho)`
5: `*Phi = FFT_SOLVE(*Rho)`
6: `*Ex, *Ey = PERIODIC_FIELD_GRID(*Phi)`
7: `t = -Δt/2`
8: `*Vx, *Vy = PUSH_V(*Ex, *Ey)`
9: **for all** `t` from `t=0` to `t=t_max` **do**
10: `*Vx, *Vy = PUSH_V(my_Np, *Ex, *Ey)`
11: `*Part_x, *Part_y = PUSH_LOC(my_Np,Vx, *Vy)`
12: `*Rho_local = PART_RHO(my_Np, *Part_x, *Part_y)`
13: `*Rho_global = MPI_Allreduce(*Rho)`
14: `*Phi = FFT_SOLVE(*Rho)`
15: `*Ex, *Ey = PERIODIC_FIELD_GRID(*Phi)`
16: `t += del_t`
17: **end for**

As we see, this is very similar to algorithm 6 on page 20. The difference is the adding of lines 4 and 13, to find the global result for `*Rho`. We also see that all the particle-

bound functions use `my_Np` instead of `Np`. This should give them all approximately linear speedup. However, the cost is the time taken for the `MPI_Allreduce()` calls.

The grid solving phases above should of course be done with with parallel solvers, even if that was not done in the code later, as time was not found to implement it.

4.1.3 A summary of the Lagrangian approach

The Lagrangian approach seems to be a good choice in certain select cases. If one has a serial PIC code that is to be parallelised, it can be implemented with small effort. However, the resulting code might not scale well, but if the point is to exploit some small-scale parallel hardware one has available, that is maybe not of importance.

However, parallel applications written from the bottom up, should be made as scalable as possible, in which case using the `MPI_Allreduce()` function is a bad idea.

Thus, in the hunt for scalability, we need another decomposition.

4.2 EULERIAN DECOMPOSITION

In 2.2 some points on how to go about supporting an Eulerian decomposition was listed. We will recap this now, grouping together the needs together by function:

- Factorise the number of processors in two, determining the number of processors in x and y.
- For each process, determine its domain size in grid points, stored in `my_Nx` and `my_Ny`. To know where in the global grid this belongs we record the grid coordinates of the lower left corner in `llc_x` and `llc_y`. The rank of the neighbouring processes are stored in `int neighbours[8]`. Boundaries for particle migration are calculated.
- All particles must be migrated to the process governing the area they reside in, if they are not within the boundaries of the process currently handling them.
- Border values must be communicated between neighbours, both at the beginning and end of the grid computations. The arrays for grid data are increased by two in each direction to keep track of these border data.

To this end, 3 functions were constructed, corresponding roughly to the different requirements outlined above.

4.2.1 `eulersplit()`

This function takes the global system sizes as input, and calculates the process' domain size, location and neighbours based on the process' rank. The function is called by all the processes, as it bases all decisions on the global parameters and process rank no communication is needed.

4.2.2 `borderexchange()`

This function will extract the ghost-point borders of the `*Rho` array, exchange the ghost points with the appropriate neighbour as found in the `neighbours[]` array, and apply the received values onto the edges of the process' own `*Rho` grid.

Algorithm 9 `eulersplit(...)`

Require: my_rank, num_procs, Nx, Ny, Lx, Ly

1: Factorise num_procs
2: `myrank_x`, `myrank_y` = location in grid
3: Determine `neighbours[]`
4: `my_Nx`, `my_Ny` = size of my subgrid in X and Y
5: `llc_x`, `llc_y` = location of my lower left corner in global grid
6: Determine grid boundaries
7: **Return** `myrank_x`, `myrank_y`, `neighbours[]`, `my_Nx`, `my_Ny`, `llc_x`, `llc_y`, boundaries.

Algorithm 10 `borderexchange(...)`

Require: `*Rho`, `*neighbours`, `my_Nx`, `my_Ny`

1: Extract borders from `*Rho`
2: **for all** Neighbours **do**
3: Exchange border points
4: **end for**
5: Apply recieved border points to `*Rho`
6: **Return** `*Rho`

4.2.3 *migrate()*

This function undertakes the particle migrations. It rounds up the out-of-bounds neighbours, and then sends them to the appropriate neighbours in a round-robin fashion. All incoming particles are added to the local list of particles, and the number of local particles are adjusted accordingly.

Algorithm 11 `migrate(...)`

Require: `Np`, `*neighbours`, `*Part_x`, `*Part_y`, boundaries

1: **for all** Neighbours **do**
2: Round up all particles now in the neighbour's domain in a buffer, removing them form `*Part_x` & `*Part_y`. Decrease `Np` accordingly.
3: Send particles, and recieve new ones.
4: Add new particles to `*Part_x` & `*Part_y`, increasing `Np` accordingly.
5: **end for**
6: **Return** `*Part_x` & `*Part_y`

4.3 ADAPTIVE EULERIAN DECOMPOSTION

An adaptive, Eulerian decomposition, we recall from ch. 2 to chapter here] is where we tune an Eulerian decomposition at certain time intervals to combat load imbalance. When all the logistics, for lack of a better word, needed for an Eulerian decomposition is in place, all we then need is an algorithm calculating a new, load balanced grid division. Then we can run through the points of listed in the Eulerian implementation description, and we are ready for a new iteration.

Chapter 5

Results

5.1 Timing `MPI_ALLREDUCE()`

First, in order to shed some light on how long the `MPI_Allreduce` takes on the cluster and on a SGI Altix machine, the test of `PART_RHO` with `MPI_Allreduce()` (see algorithm 7 on page 22) was used. The timings of the `MPI_Allreduce()` call was done with the `MPI_Wtime()` function.

The first round of timings, shown in 5.1 were done by keeping the number of grid points fixed, but varying the number of nodes from 1 to 8. As we see, on ClustIS the time increases quite dramatically. The SGI Altix machine, however, shows only a slight increase. The "step like" nature of the graph (especially visible on ClustIS), might be due to the tree-nature of the Allreduce function, as all increases in the number of processors does not equate to an increase in tree depth, necessarily.

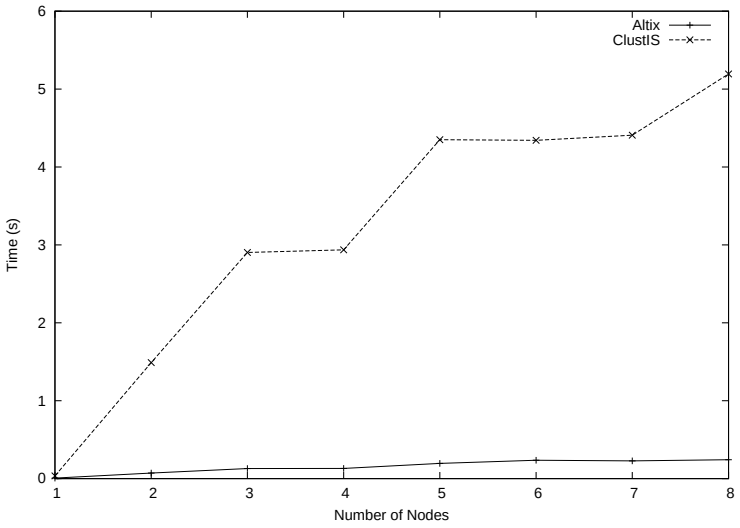

Figure 5.1: MPI_Allreduce timings, with one million grid points.

The situation with 4 and 8 nodes were inspected a bit more thoroughly. Figure 5.2 show how increasing the number of grid points, that is the amount of data to communicate, impacts on the run times. As we see, the amount of data impacts quite heavily,

which is only to be expected. Once again, we see that the Altix machine is in another division with regards to communication. Note that the X-axis on the graphs are logarithmic.

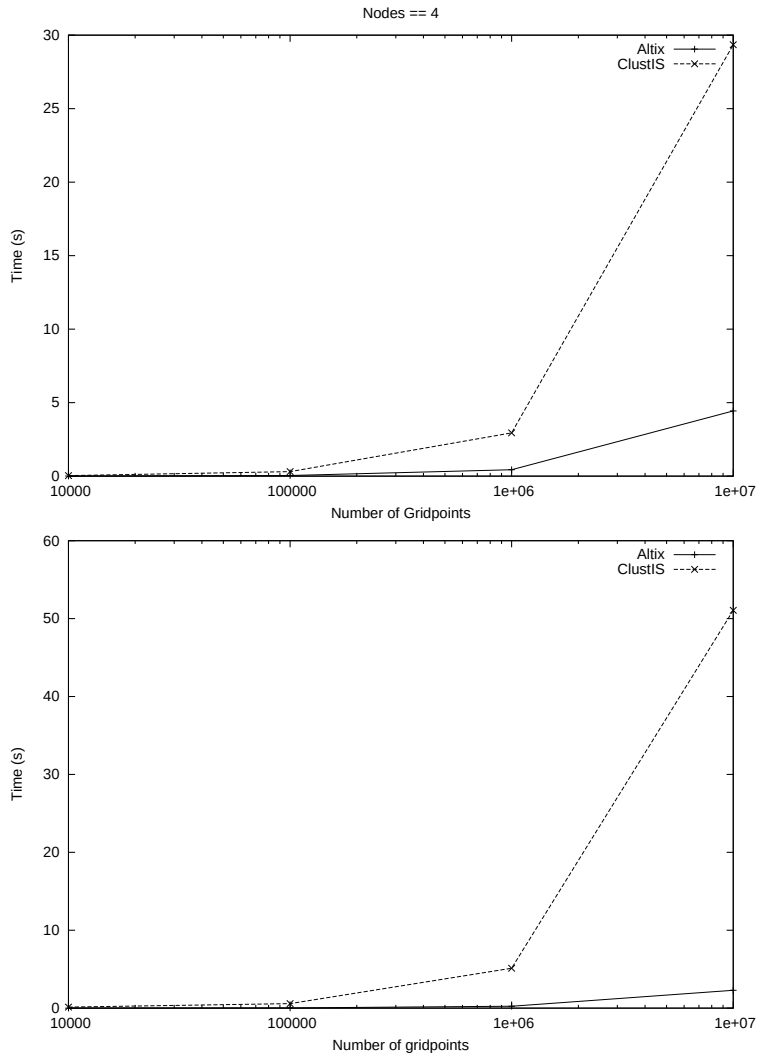

Figure 5.2: MPI_Allreduce times on ClustIS and Altix with 4 and 8 nodes

What we can see from this graph is that up to a 100000 grid points, the Allreduce takes a very short time on both ClustIS and the Altix machine. At 1 million, it is still very small on the Altix, but on ClustIS it is taking several seconds. However, we see that for PIC codes intended for use on small-scale clusters, the Allreduce is not prohibitively expensive.

5.2 LAGRANGIAN SIMULATION

The test here is conducted with code as outlined in algorithm 8 on page 22. It was run with a $128 * 128$ grid, with 1048576 (2^{20}) particles.

5.2.1 *ClustIS*

The tests was done using from 1 to 19 processors/nodes. The results are rather as expected. As ClustIS is a multiuser, multitasking system, a process never has the system entirely alone. This probably makes up for the small variations we can see in the timings. However, the grid solving phases are more or less using a constant amount of time, as expected, since they are not parallelised. The other computational parts are exhibiting a close to linear speedup. The curve for `MPI_Allreduce` shows how its execution time climbs as the number of nodes increases. The slight stepwise nature of the curve is explained by the tree-structure-based nature of the function. Its depth is log_2 of the nodes, which explains the "steps" we see in the graph.

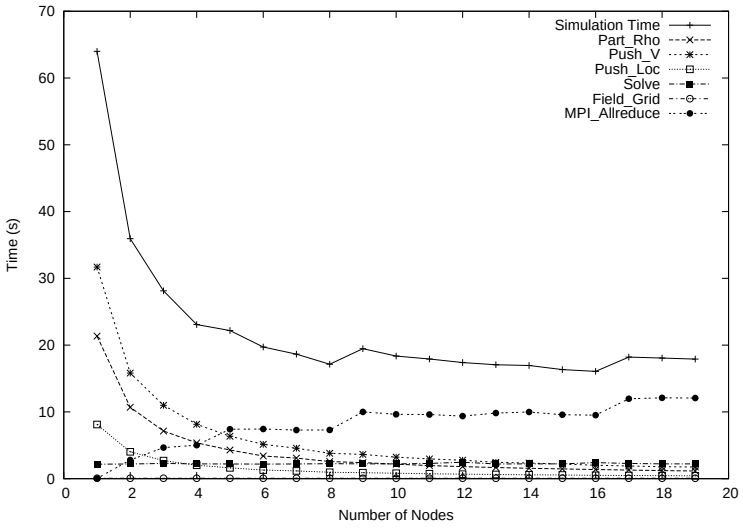

Figure 5.3: Timings of Lagrangian run on ClustIS

5.2.2 *SGI Altix*

First, a word about compilers. When the PIC code was copied to the Altix, the GCC compiler was used, as it's the same compiler that is used on ClustIS. The communication speed was impressive, but the computation times was off the chart! The computational parts took about *three* times as long as on ClustIS. Looking around on the machine, an optimised Intel compiler was found. Binaries from the exact same C code executed 5.8 times faster with this compiler. Now the Altix revealed itself as almost twice as fast on a processor-to-processor-level compared to ClustIS. At first, this was thought to be caused by the GCC compiler generating IA32 code. This would force the Intel Itanium 2 processors to run in 32-bit backwards compatibility mode, which is not optimal. However, checking with the Unix `file` command, it was revealed that GCC had in fact generated a 64-bit executable. In other words, the Intel compiler produces vastly

superior machine code for the Itaniums. The moral must be to not skimp on compilers when using Itaniums, at least not until the GNU compiler is vastly improved.

Now for the test results. The test is the same as on ClustIS, $128 * 128$ grid and 2^{20} particles. As we see, the curve slopes down slightly more smoothly than it did on ClustIS. If we look at the line for `MPI_Allreduce` we see the reason - the jump in its execution time we had especially from 4 to 5 nodes on ClustIS is undetectable. However, if we look closely at the total time curve, we see it edge ever so slightly upwards from seven to eight nodes. Looking at the `MPI_Allreduce` time, we see the culprit. So not even "super-class" communication saves us entirely from the cost of the Allreduce. However, the increase in communication cost is very small, so if if we had a simulation with more particles (say tens of millions), it would not make much of an impact. Sadly, the Altix machine has only eight nodes, so how SGI's NUMAFlex communication architecture would handle larger Allreduce could not be inspected.

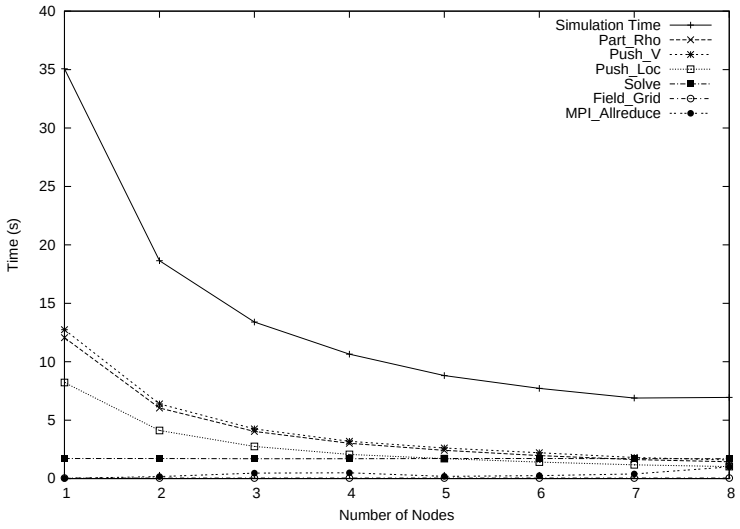

Figure 5.4: Timings of Lagrangian run on the Altix

5.3 EULERIAN SIMULATION

Sadly, a full Eulerian simulation was never assembled. The main reason for this was again the lack of a proper output system. As both the grid and particles are scattered across processors, it is hard to make some sort of ad hoc way of extracting useful information. Even tracing the particles is a challenge, as they would have to be tracked across processes on different machines. This would call for some way of labelling particles, so that they can be individually identified, which would require some extra modifications of the code.

In addition, several of the numerical functions would need to be modified, as border conditions are implemented directly into them. In an eulerian scheme, this must be handeled by special functions, as border conditions depend on where in the grid one are. For grid borders this means that `borderexchange()` must handle it, for particles it must be handeled in `migrate()`. Both these functions will apply periodical borders. The numerical functions, in their current state, will be confused by the fact that they

only see the local sub grid, and thus enforce periodical borders on it. As the eulerian scheme outlined calls for the addition of an extra border of grid points on all grid arrays, the modification needed in the numerical functions is simply to make them disregard border cases and treat everything as interior (of course stray particles will need to be caught to avoid segmentation fault).

All the Eulerian sub-functions were properly tested, and they all work. So once HDF support is in place, it should be quick to add the calls to the functions into the

Chapter 6

Conclusion & Future work

As we can see, programming PIC codes for Clusters with MPI gives good performance.

We also saw how impressive true "Supercomputer-grade" communication can be. But we allso see that though fast, it's still a cost to be reckoned with. Adding to that the total cost of an SGI machine compared to a Cluster, it is understandable that clusters are making inroads on the HPC scene.

Also note the extreme performance differences between the GNU compiler's code and the Intel Compiler - the Intel's code 5.8 times faster! So for at the time of this writing, using Intel's own compilers for the Itanium processor is highly recommended.

One thing lacking here, is the use of the code to simulate some more sophisticated plasma phenomena. The reason for this is twofold. First, the author has very little knowledge of plasma physics, and as such were not able to make any good scenarios for simulation. Second, the code is lacking in the input/output department, thus making it hard to actually look at the simulation results. Whilst the correctness of the code was checked by tracking a few particles and check if they exhibited plasma oscillations, that is hardly useful in order to look for more sophisticated phenomena.

The best approach would probably to look into the HDF format, and install it on ClustIS and other computers to be used. The HDF format is built to support parallel, scientific applications, and has a lot of very convenient features that would make it reasonably easy to provide a coherent output even from the Eulerian version, which is a challenge (easy once one has learned to use parallel HDF, that is). This is because it supports writing sub grids (called "hyper slates") in parallel into one file. The files written with Parallel HDF are readable by serial HDF applications. Which makes it perfect to solve the output challenges an Eulerian decomposition poses.

Further the code should be extended to 3 dimensions. This will, however, pose even more challenges in the output department, so it might be a good idea to have that sorted before embarking on that extension.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Erol Akarsu, Kivanc Dincer, Tomasz Haupt, and Geoffrey C. Fox. Particle-in-cell simulation codes in High Performance Fortran. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press. ACM Order Number: 415962, IEEE Computer Society Press Order Number: RS00126.

[2] Anne C. Elster. Software test-bed for large parallel solvers.

[3] Anne C. Elster. Parallelization issues and particle-in-cell codes, August 1994.

[4] R.A.Fonseca, L.O.Silva, R.G.Hemker, F.S.Tsung, V.K.Decyk, W.Lu, C.Ren, W.B.Mori, S.Deng, S.Lee, T.Katsouleas, and J.C.Adam. Osiris: a three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators, 2002.