A High-Performance 1D1V Electrostatic Particle-in-Cell (PIC) Code

Matt Russell

(Dated: December 4, 2023)

Modern plasma experiments require numerical simulation to be performed in order for their findings to be properly understood. These computer experiments fall into one of two basic categories, depending on what perspective they consider the plasma from. Kinetic simulations, the first kind, view the plasma as a collection of discrete, microscopic, "superparticles". This article describes the basic algorithm behind a kinetic plasma simulation, and explains how to write a high-performance PIC code, which can be found on Github.

I. Introduction

Plasma as an N-body Problem

A plasma is a collection of charged particles which exhibit collective effects. If this ensemble of electrically-charged constituents is considered to be in thermodynamic equilibrium, then it can be approached from the perspective of behaving like an electrically-conducting fluid via the field of Magnetohydrodynamics (MHD).

However, many important phenomena in plasmas are inherently non-equilibrium, and furthermore, many of the "equilibrium" found in a plasma are really quasi-equilibria, i.e., quasi-steady states. These can only be approximated fairly by fluid behavior, not described exactly. An exact description of a plasma is equivalent to that of solving an N-body problem for $N \sim 10^{20}$ charged particles, which are each subject to the influence of the electromagnetic fields consistent with those being radiated away by the others.

This problem is, unfortunately, intractable, i.e., unsolvable. It would necessitate solving a differential equation for each particle's motion, and resolving the electromagnetic interaction from its $N-1$ fellows in the process,

$$\vec{F}_i = m_i \frac{\mathrm{d}^2 \vec{x}_i}{\mathrm{d}t^2} = q_i \left(E_i + \vec{v}_i \times \vec{B}_i \right) \tag{1}$$

$$\vec{E}_i = \sum_{i \neq j} \vec{E}_{ij} \tag{2}$$

$$\vec{B}_i = \sum_{i \neq j} \vec{B}_{ij} \tag{3}$$

In the microscopic case, the electromagnetic fields could be computed by adding up the contribution, e.g., using the Lienard-Wiechert fields, of all the other particles in the system.

Not only is this atomic computation extremely expensive, with the Lienard-Wiechert potentials being given by,

$$\phi(\vec{r}, t) = \frac{1}{4\pi\epsilon_0} \left(\frac{q}{(1 - \vec{n}_s \cdot \vec{\beta}_s)|\vec{r} - \vec{r}_s|} \right)_{t_r} \tag{4}$$

$$\vec{A}(\vec{r}, t) = \frac{\vec{\beta}_s(t_r)}{c} \phi(\vec{r}, t) \tag{5}$$

$$\vec{\beta}_s(t) = \frac{\vec{v}_s(t)}{c} \tag{6}$$

$$\vec{n}_s = \frac{\vec{r} - \vec{r}_s}{|\vec{r} - \vec{r}_s|} \tag{7}$$

but, every timestep of this hypothetical computation, they would need to be computed $N-1$ times, for *each* of the N particles, giving a work of $\mathrm{O}(N^2)$, where $N \sim 10^{20}$.

This is simply too much work for even the most powerful computer imaginable to perform.

II. The Particle-In-Cell (PIC) Algorithm

Instead of engaging in the fool's errand of trying to solve such an N-body system directly, physicists developed the field of statistical mechanics in order to have a tool with which they could make predictions about the behavior of these kinds of very large systems, by computing their macroscopic properties.

In the 1950s, and 60s, computational scientists in America and the United Kingdom began to develop methods for the simulation of particle systems. These methods work by advancing the state of the particles on a grid, via the solution of the self-consistent electromagnetic problem which determines the electric and magnetic fields on the grid based on the charge, and current density of the particles weighted to it. In the modern day, we refer to these methods under the umbrella of "Particle-in-Cell" (PIC) algorithms.

The PIC algorithm is visualized below in Figure (1)[1]. This loop continues until a sufficient time has been reached that the simulation halts. Extra steps for initializing and writing the data out can be inserted at their appropriate places in the graph to give a full picture of the simulation.

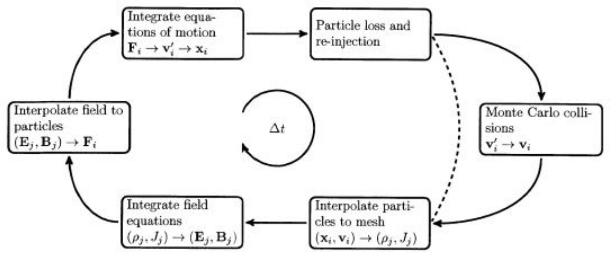
Integrate equations of motion $\mathbf{F}_i \to \mathbf{v}'_i \to \mathbf{x}_i$
Particle loss and re-injection
Monte Carlo collisions $\mathbf{v}'_i \to \mathbf{v}_i$
Interpolate particles to mesh $(\mathbf{x}_i, \mathbf{v}_i) \to (\rho_j, J_j)$
Integrate field equations $(\rho_j, J_j) \to (\mathbf{E}_j, \mathbf{B}_j)$
Interpolate field to particles $(\mathbf{E}_j, \mathbf{B}_j) \to \mathbf{F}_i$
Δt

FIG. 1. Graph of the flow of a timestep in the full, electromagnetic, PIC algorithm

III. 1D1V Electrostatic PIC

The complete theory of a PIC simulation can be found in either the writings of Langdon and Birdsall[2], or Hockney and Eastwood[3], all of which were pioneers in the field. Instead of trying to reinvent what they have already done, this article will focus on describing the implementation and results of a high-performance 1D1V electrostatic (ES) PIC code developed by the author.

'1D1V' refers to a phasespace composed of only a single spatial dimension, and single velocity dimension. In this configuration, the equation of motion of the particle's collapses into a scalar form, and the task of solving Maxwell's equations greatly simplifies,

$$\vec{F}_i = m_i \frac{\mathrm{d}^2 \vec{x}_i}{\mathrm{d}t^2} \to F_i = m_i \frac{\mathrm{d}^2 x_i}{\mathrm{d}t^2} \tag{8}$$

$$\nabla \cdot \vec{E} = \frac{\rho}{\epsilon_0} \to \frac{\mathrm{d}E}{\mathrm{d}x} = \frac{\rho}{\epsilon_0} \tag{9}$$

These are the basic relevant equations for a 1D1V PIC code, and furthermore, we would need additional velocity dimensions before having to consider the magnetic field (j_x does not provide support for a B_x), or inductive electric field. As a consequence of the electrostatic character of the fields, we can write

$$E = -\frac{\mathrm{d}\phi}{\mathrm{d}x}$$

and the task of solving Maxwell's equations becomes that of solving Poisson's equation in 1D.

A. Particle Weighting

Each simulation particle has a given position, x_i. In a 1D1V simulation, this information is used to determine the charge density, ρ_j, on the gridpoint, x_j, based on the shape, $S_m(x)$, that we choose for the particles. The simplest, sensible choice of shape that can be made is that of a uniform charge cloud with a diameter a_0, and height h_0.

This "charge cloud" represents the 0th-order shape for the particles, and the origin of the original, "Cloud-in-Cell" name for the algorithm. This signal, shown in Figure (2), is, of course, just a unit pulse,

$$S_0(x) = \begin{cases} h_0, & -\frac{a_0}{2} < x < \frac{a_0}{2} \\ 0, & else \end{cases}$$

The amplitude of the signal, which represents the charge density of a particle, is chosen based on the requirement that charge be conserved in the system. Mathematically, this is expressed by integrating the charge density over the domain,

$$Q = \int \sum_i q_i S_m(x) dx \tag{10}$$

$$= \sum_i q_i \int S_m(x) dx \tag{11}$$

$$\therefore \int S_m dx = 1 \tag{12}$$

from which we can determine that $h_0 = \frac{1}{a_0}$ satisfies this.

The simplest way of apportioning the charge is to find

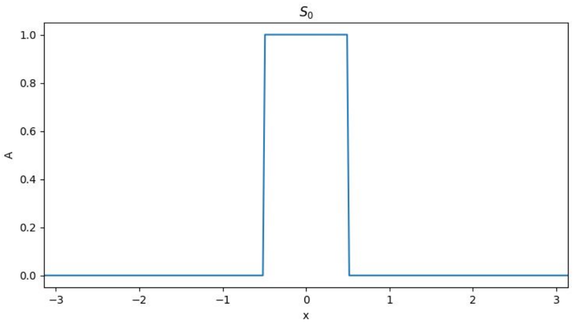

FIG. 2. 0th-order shape function on the domain $x \in [-\pi, \pi]$, with $a_0 = 1.0$.

which gridpoint is nearest the particle's location, x_i, and place all the charge there. However, this produces noisy charge distributions as particles will suddenly 'leap' from one point to the other. To address this, we can refine the particle shape and consider a piecewise function in the shape of a triangle,

$$S_1(x) = \begin{cases} \frac{h_1}{a_0}(x + a_0), & 0 < x < a_0 \\ \frac{h_1}{a_0}(x - a_0), & -a_0 < x < 0 \end{cases}$$

This shape is not chosen arbitrarily. Higher-order shape functions are computed based on a convolution of the previous order,

$$S_m(x) = S_{m-1}(x) * S_{m-1}(x)$$

The height, $h_1 = \frac{1}{2a_0}$, can be computed from the requirement that the shape function be normalized.

Once the particulars of the particle shape are decided upon, the cell that the particle is located in must be found. In practice, particle shapes that are not 1st-order are typically not chosen. Higher-order shapes converge to a Gaussian, and the 0th-order "cloud" introduces too much noise into the simulation to be practically sensible.

After finding the cell in which the charge is located, its density is apportioned to the neighboring grid points using a method of inverse distances,

$$\rho_{c,j} = q_i \frac{a}{\Delta x} \quad (13)$$

$$\rho_{c,j+1} = q_i \frac{b}{\Delta x} \quad (14)$$

$$a = |x_{j+1} - x_i| \quad (15)$$

$$b = |x_i - x_j| \quad (16)$$

For a 1D1V electrostatic simulation, this is all that is required. More complicated programs involving magnetic fields will require that the current be weighted as well, and 2D / 3D codes will require that inverse areas or volumes used when weighting the particles.

B. Field Solve

In general, the purpose of the previous, particle-weighting step is to obtain values for the charge and current density so that Maxwell's equations can be solved to obtain the electric and magnetic fields on the grid. In an electrostatic, 1D1V simulation this reduces to the task of solving Poisson's equation for the electric potential,

$$\frac{\mathrm{d}^2\phi}{\mathrm{d}x^2} = -\frac{\rho_c}{\epsilon_0} \quad (17)$$

To achieve this, the second derivative can be discretized using a finite (central) difference, and the resulting system expressed as a collection of linear equations,

$$\frac{1}{\Delta x^2}\left(\phi_{j+1} - 2\phi_j + \phi_{j+1}\right) = \frac{\rho_j}{\epsilon_0} \quad (18)$$

To solve this, the system can be formulated as a matrix and then suitably inverted, or it can be solved iteratively. If neither of these approaches are satisfactory, it can also be solved via Fourier transform.

C. Force Weighting

After the particles have been accumulated on the grid, and the electromagnetic fields obtained, their impact on the particles must be computed. This is done by weighting the fields on the grid back to the particles, similar to how the charge and current density of the particles was originally weighted to the grid.

1. 0^{th}-order Weighting

At the simplest level, the nearest grid point (NGP) method selects the closest grid point to the given particle, and assigns its electric field to the particle.

2. 1^{st}-order Weighting

One level up is to use the method of inverse distances again. Inverse distances are important as computations using them result in the conservation of densities, e.g, charge or momentum, without introducing the noise of a 0th-order method.

$$E_i = q_i \frac{a}{\Delta x} E_j + q_i \frac{b}{\Delta x} E_{j+1} \quad (19)$$

the above expresses the force on the particle, i, that is located in cell j.

D. Particle Push

After the forces being felt by the particles have been computed, the particle motion must be advanced. In order to obtain second-order accuracy a Leapfrog method is employed.

1. Leapfrog Method

The Leapfrog method advances the position and velocity of a particle subject to an external force by calculating the velocity at 'half-steps', and then using this to advance the position,

$$v_i^{n+1/2} = v_i^{n-1/2} + \Delta t * E_i \quad (20)$$

$$x_i^{n+1} = x_i^n + \Delta t * v_i^{n+1/2} \quad (21)$$

while being very accurate, the Leapfrog method also suffers from an instability that leads to phase-error if the condition $\omega\Delta t << 2$ is not satisfied.

The flow of each timestep is visualized in Figure (3), showing the relationship between the simulation variables of an electrostatic, 1D1V, PIC code.

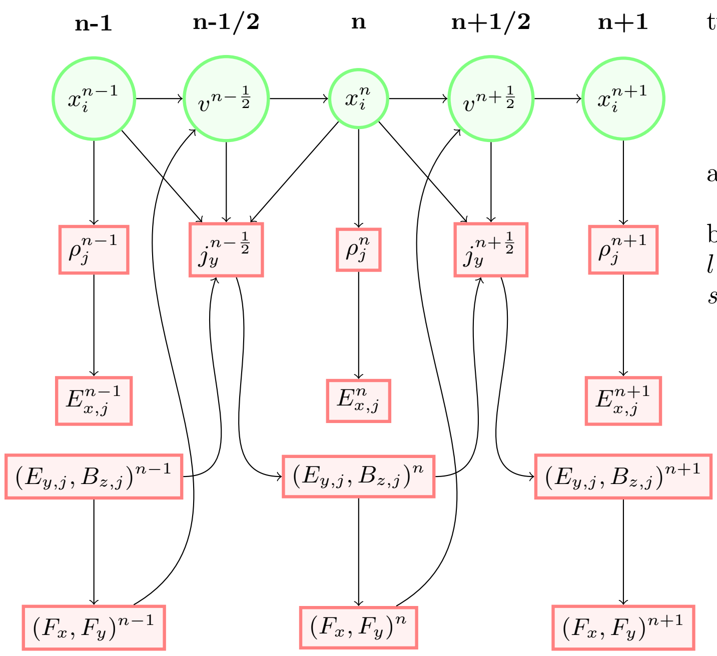

FIG. 3. The directed graph (digraph) describing the dependency among the simulation quantities at each timestep. The direction of the arrows indicates which data are used in the computation of the variables being pointed to. Green boxes represent a particle quantity, and red boxes represent an electromagnetic quantity.

IV. Simulation - How to Create a High-Performance Scientific Code

Plasma science is the union of plasma theory, experiment, and simulation. This statement might be trivial, but the complexity of the systems that plasma science studies are certainly not. Gaining insight into the behavior of a fusion device, or astrophysical structure, requires more than just the approximate (albeit powerful) tools of plasma physics. The limited nature of plasma diagnostics, typically only able to resolve quantities along a given direction or in an average-sense, poses further difficulties.

High-performing software, capable of processing large volumes of data in relatively short periods of time, addresses these challenges. The key to writing such a scientific code is parallelizing it. However, a parallel code will only be as fast as the serial code it is based on allows it to be, and there are many pitfalls to building a fast parallel code that an unsuspecting developer can shoot themselves in the foot with, e.g., race conditions, and inefficient communication, to name a couple. Therefore, the correct approach is to begin with a fast serial code.

A **fast serial code** begins with the right **data structures**, and **algorithms** for the job. Anything beyond this is a function of the hardware the code is running on. As far as a PIC code is concerned, the algorithm we must implement is largely specified for us, however, there are two key locations where we have the freedom to choose, and making the right choice represents the difference between our program being fast, or slow. Every timestep, two things ***must*** occur,

1. Finding the particles
2. Solving the fields

and this is where we can make our hay.

Notice that the grid in which the particles are going to be searched for is *ordered* (ascending), i.e., $x_j > x_l \quad \forall j > l$. This is important because it means we can use a *binary search* for finding the particles.

A. Binary Search

B. Sparse Solve

C. Non-dimensionalization

D. C++

C++ is generally considered the programming language of choice for high-performance scientific code. In recent years, challengers such as Rust and Julia have emerged, but the simple truth is that none of these languages (C++ incl.) will perform better than C.

However, instead of writing everything in C, C++ is typically chosen because it gives a programmer access to the performance of C, at the cost of a slight reduction to the program's wall time because of the abstraction that goes into masking the underlying C. Of course, C has its own drawbacks, namely, memory safety, and developer time. Hence, why C++ is chosen, and C is left for the parts of a code that really need the speed.

Currently, only a C++ version of the code has been implemented, as that is what the author knows techniques for writing fast at this moment in time.

V. Results

Once a scientific code has been written, it must be **benchmarked** to ensure that it achieves the performance necessary for the target applications, and it must be **validated** to ensure this performance is meaningful.

Benchmarking is the task of measuring the speed of a program. The most meaningful performance metric is the **wall-time**, the time it takes for the program to halt, and second to this is the rate at which the code performs ”floating-point” operations, termed **Flops**.

Speed is meaningless if the problem is being solved incorrectly, so scientific (all) code must also be validated to make sure this is not the case.

Benchmarking

A sweep of (N, N_x) parameter-space was automated, and performed over the course of three days, in order to determine the performance of the simulation kernel, i.e., the PIC loop minus writing the data out, that spanned all values possible on the target hardware. The performance of writing the data out will depend largely on whether the storage device that the data files are being stored on is a solid-state disk, or a magnetic hard-drive, so only the compute kernel is benchmarked in order to isolate the performance of the algorithms. The results of this large, automated, computation are shown in Figure (4).

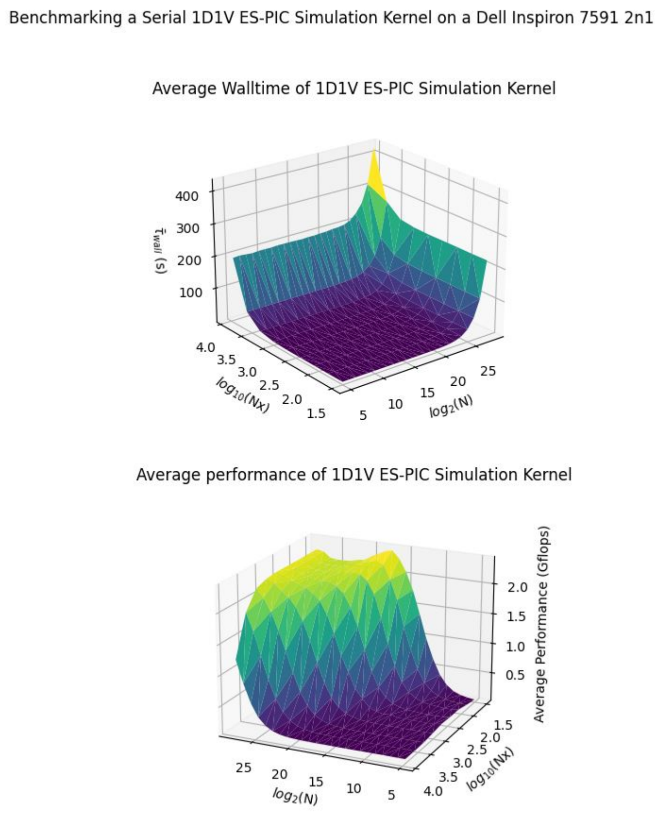

FIG. 4. Performance of the simulation kernel across the domain $N \times N_x$ with $N \in [2^5, 2^{28}], N_x \in [2^5, 2^{13}]$. This spans the complete set of values that it is possible to run on the Dell Inspiron 7591 2n1 which was processing the data. The complete dataset can be found in Appendix A.

Performance metrics were obtained using the bash command "perf stat -e r5301c7 ./bench", where the "r5301c7" string corresponds to the hardware code on the target hardware which counts the number of floating point operations performed using the execution of the binary file "bench". The author constructed a pipeline that automated the gathering of this data so that adequate statistics could be taken. It was necessary to determine the exact amount of arithmetic being performed as the binary was compiled with the maximum compiler optimizations, making estimates of the number of operations from the program text inaccurate.

Validation

Computational plasma physics has a host of classic problems for validating the operation of a PIC code.

Electron Motion

When electrons in a plasma feel an electric field, their small inertia means they respond to the impulse on a timescale that is shorter than any other possible timescale in a plasma. Real plasmas consist of many electrons, most commonly thermalized, and part of the Debye cloud of a positive ion. In the rest frame of the electron, the electric field that is set up by the space charge of this ion will continuously act to pull the electron inwards, much like a satellite orbiting the Earth continues to fall, except much faster and smaller scale.

Langmuir Oscillations

Single-particle motions are only one aspect of basic plasma physics. Collective effects matter a great deal as well. Just as the previously-discussed electron feels an influence from the positive ion whose Debye cloud it is a member of, so too does it exert an influence on its fellow electrons. As the negatively-charged particle oscillates, it will do so as a part of a collective oscillation of its species.

Two-stream Instability

The research literature of plasma physics is primarily concerned with the topic of **instabilities**, i.e., disturbances in a plasma which possess a finite lifetime, and disrupt its prevailing state of equilibrium, often in a violent fashion. This prevailing interest stems from the flagship program of plasma physics, namely that of igniting controlled nuclear fusion.

The canonical example of an instability is that of the *two-stream instability*. When two counter-streaming beams (currents) of electrons flow past each other, the magnetic force felt by one beam from the motion of the other, and vice-versa attracts the two together. If the currents were bound to the interior of a pair of conducting wires, then this would be exactly the same situation as is observed in an introductory magnetism course when a current is run in opposite directions, through parallel wires.

However, all the charges in a fully-ionized plasma, which a fusion plasma surely is, are free. There is no work function holding them in place, so instead they will begin to mix, forming islands in phase-space as the particles stream past.

VI. Conclusion and Future Work

A high-performance 1D1V PIC code is not an easy task to write, but it is also only the beginning. There are three main thrusts in which future work will take place.

1D2V

A 1D1V code is the most basic kind of PIC algorithm that can be implemented. The most advanced code possible is a full 3D3V electrodynamic algorithm which solves the full set of Maxwell's equations in space-time, and evolves the particle's motion in 6D phase-space. The leap from a 1D1V code to a 3D3V one is significant, and the minimum relevant configuration for an electromagnetic PIC is 1D2V, so this should be implemented first. This problem incorporates a current in the y-direction, j_y, which supports a magnetic field in the x-direction, B_x.

Parallelization

The computation to sweep the simulation kernel for performance metrics took over three days of straight work to complete. Even though the end-to-end simulation was not benchmarked for logical reasons, this is still a task that should be done in order to understand the application's performance when deployed to a platform with a suitably fast storage device. To speed up this process, the simulation should first be parallelized so that multiple threads can work concurrently to complete the sweep in a shorter timeframe.

Parallelization should also be implemented as a matter of course. Even the fastest serial code cannot hope to compete with a correctly-programmed parallel code, and it is the centerpiece of a high-performance computational scientist's toolbox. However, parallel code is not easy to write. Data dependencies, race conditions, and handling multiple threads necessitates more complicated code, which in turn makes debugging more difficult. Furthermore, multiple threads are not a free lunch for speed, and they require an efficient communication algorithm to maximize their potential, and achieve the performance gains that proper high-performance code should.

A Competition Among Programming Languages

C++ was invented over 40 years ago. Numerous C++ standards in the interim attempt to modernize the language, but with new challengers, e.g., Rust and Julia, as well as the granddaddy of them all, C, it is worth seeing how they stack up against each other, and which can write the fastest 1D1V PIC code. In this vein, it would be an insightful, and fun, exercise to implement versions of the 1D1V code in these languages, and benchmark their performance. Additionally, due to the ubiquity of Python in computational science, and the venerable history of Fortran, these languages should also be "thrown into the ring", so to speak.

[1] J. Estes, Implementation and performance evaluation of a gpu particle-in-cell code (2012).

[2] C. Langdon and A. Birdsall, *Plasma Physics via Computer Simulation* (CRC Press, 1991).

[3] R. Hockney and J. Eastwood, *Computer Simulation Using Particles* (Routledge, 1989).

Appendix A: Benchmarking data

PIC Kernel