Journal of Computational Science 27 (2018) 345–356

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Efficient data layouts for a three-dimensional electrostatic Particle-in-Cell code

Yann Barsamian [a,b,*], Sever A. Hirstoaga [b,c], Éric Violard [a,b]

[a] Université de Strasbourg, CNRS, ICube UMR 7357, F-67412 Illkirch, France
[b] Inria Nancy - Grand Est, F-54600 Villers-lès-Nancy, France
[c] Université de Strasbourg, CNRS, IRMA UMR 7501, F-67084 Strasbourg, France

ARTICLE INFO

Article history:
Received 10 December 2017
Received in revised form 22 April 2018
Accepted 10 June 2018
Available online 26 July 2018

Keywords:
Data structures
Space-filling curves
SIMD architecture
Hybrid parallelism
Strong and weak scaling
Three-dimensional Particle-in-Cell simulation
Plasma physics

ABSTRACT

The Particle-in-Cell (PIC) method is a widely used tool in plasma physics. To accurately solve realistic problems, the method requires to use trillions of particles and therefore, there is a strong demand for high performance code on modern architectures. The present work describes performance results of `Pic-Vert`, a hybrid OpenMP/MPI and vectorized three-dimensional electrostatic PIC code.

The code simulates 3d3v Vlasov–Poisson systems on Cartesian grids with periodic boundary conditions. Overall, it processes 590 million particles/second on a 24-core Intel Skylake architecture, without hyper-threading (25 million particles per second per core).

The paper presents extensions in 3d of our preliminary 2d results (Barsamian et al., 2017), with highlights on the difficulties and solutions proposed for these extensions. Specifically, our main contributions consist in proposing a new space-filling curve in 3d (called L6D) to improve the cache reuse and an adapted loop transformation (strip-mining) to achieve efficient vectorization. The analysis of these optimization strategies is performed in two-stages, first on a 24-core socket and second on a super-computer, from 1 to 3072 cores, demonstrating significant performance gains and very satisfactory weak scaling results of the code.

1. Introduction

Implementing a Particle-in-Cell (PIC) method is an important step of many applications in the field of computational science. Although particle methods are usually characterized by low accuracy, they are able, when using a very large number of particles, to adequately reproduce complicated phenomena, for instance in plasma physics [2,3]. However, in such a situation, a naive implementation can quickly exhibit memory bottlenecks since the overall cost is dominated by data motion and not by computation [4].

Therefore, a lot of research efforts were recently devoted towards more adapted PIC implementations that utilize efficiently modern super-computing resources [4–9]. Despite these significant advances, targeting realistic applications in plasma physics is still challenging, due to the complex multi-scale six-dimensional problems to be solved in the phase space. In this direction, our objective is to develop a massively parallel and highly scalable PIC code, in view of physically meaningful realistic simulations.

First steps were achieved in [10,1] where we built a hybrid parallel and vectorized PIC code for a Vlasov–Poisson model in a two-dimensional (2d) physical space. Several data structures for particles and for grid quantities were analyzed in order to enhance data locality and to reduce the execution time with respect to a classic code. More precisely, we showed in [1] that a structure of arrays and a L4D space-filling curve lead to an efficient ordering of the particles and of the electric field and the charge density, respectively. In addition, the performance of parallelization of the loops through distributed and shared memory paradigms was assessed in tandem with the memory channels.

In the present contribution we extend to three-dimensions (3d) for the physical space the PIC code in [1] for simulating electrostatic plasma. Even though reduced 2d models are used in the literature to gain insights about the main behavior of the plasma, full 3d simulations clearly improve the realism of the physical description. Moreover, in some situations, reducing the dimensionality is not even possible and thus a 3d simulation is unavoidable. However, with the aim of keeping a satisfactory accuracy of the computed

* Corresponding author at: Pôle API, 300 bd Sébastien Brant, CS 10413, F-67412 Illkirch Cedex.
E-mail addresses: ybarsamian@unistra.fr (Y. Barsamian), sever.hirstoaga@inria.fr (S.A. Hirstoaga), violard@unistra.fr (É. Violard).

https://doi.org/10.1016/j.jocs.2018.06.004
1877-7503/

solutions in this case, we need to consider the significant increase in the amount of data to be processed.

A first obvious difficulty of the extension from 2d to 3d simulations is the need for more storage for the grid quantities [2,3]. If, in a 2d simulation, keeping the whole grid quantities in the cache memory is still possible, this target is much more difficult or even impossible to achieve for a fine grid in the 3d case. Secondly, a 3d simulation requires more data traffic and computations. Specifically, passing from 4 grid points to 8 points for the interpolation and accumulation steps (when using a linear approach) becomes more difficult to handle, both for the data flow and for the vectorization. In addition, the difficulty increases significantly if higher order approximations within these steps are in use. The same challenges need to be addressed when vectorization is used for the updating positions step.

The paper is organized as follows: in Section 2 we present the basic kinetic model for the plasma, we detail the steps of the PIC implementation, introduce the related work and explain our contributions. In Section 3 we detail the code optimizations and we present the performance results on 24 cores with OpenMP only. In Section 4 we show the scalability of the code on up to 3,072 cores of the supercomputer Marconi. Section 5 summarizes the work and presents some future directions.

2. PIC overview for the 3d Vlasov–Poisson model

2.1. Description of the problem

A PIC method simulates a plasma by integrating self-consistently the trajectories of charged particles with fields that are generated by the particles themselves [2,3]. In the case where there is no other external field and the self-consistent magnetic field is neglected, this relies on solving the following Vlasov–Poisson system:

$$\begin{cases} \partial_t f + \mathbf{v}\cdot\nabla_{\mathbf{x}} f + \dfrac{q}{m}\mathbf{E}\cdot\nabla_{\mathbf{v}} f = 0 & \text{Vlasov} \\ f(\mathbf{x},\mathbf{v},0) = f_0 & \\ -\Delta\phi = \dfrac{\rho}{\epsilon_0} & \text{Poisson} \end{cases}$$

where

$$\rho(\mathbf{x},t) = q\int f(\mathbf{x},\mathbf{v},t)dv \quad \text{and} \quad \mathbf{E}(\mathbf{x},t) = -\nabla\phi(\mathbf{x},t).$$

In the system above, $f = f(\mathbf{x},\mathbf{v},t)$ stands for the distribution of one species of particles (with charge q and mass m) in a six-dimensional phase space (three dimensions for positions and three dimensions for velocities), ρ stands for the charge density, and $\mathbf{E}$ for the self-consistent electric field. A PIC method consists in discretizing (sampling) the distribution function by a collection of macro-particles that move in the phase space following the characteristics of the Vlasov equation. Then, a PIC simulation follows four steps: accumulate on the spatial grid the particle charge, solve the Poisson equation to obtain the grid electric field, interpolate this field to the particles, and finally push in time the particle positions and velocities.

In our PIC code, the particle positions and velocities are initialized randomly using the WELL generator [11]. The particles all have the same fixed weight. They are advanced in time with a leap-frog time stepping [2,Section 2.4] (second-order in time). The electric field is computed by solving the Poisson equation on a uniform Cartesian grid, by a Fourier method. Then, for the particle and force weighting we use the Cloud-in-Cell model [12] (first-order in space), meaning that eight neighboring grid points are used in the interpolation/accumulation steps for a particle in that cell. The PIC pseudo-code is detailed in Fig. 1.

Next we describe the two basic types of data on which the sequential implementation of the code is based.

2.2. Particle data structure

A particle is given by its position and velocity. While the velocity is classically represented by three real numbers, the position is identified in the present work with a single cell index i_{cell} and three normalized offsets within this cell. The advantages of this representation are exposed in [4,Section III-E]. In addition to the parameters explained in Fig. 1, we denote the physical space by $[x_{min};x_{max})\times[y_{min};y_{max})\times[z_{min};z_{max})$, and the grid spacing by $\Delta x = (x_{max}-x_{min})/ncx$, $\Delta y = (y_{max}-y_{min})/ncy$ and $\Delta z = (z_{max}-z_{min})/ncz$. Thus, a particle positioned at $(x_{\text{physical}}, y_{\text{physical}}, z_{\text{physical}})$ is mapped on the grid at the position $(x, y, z) \in [0;ncx)\times[0;ncy)\times[0;ncz)$, where

$$x = \frac{x_{\text{physical}} - x_{\min}}{\Delta x}, \quad y = \frac{y_{\text{physical}} - y_{\min}}{\Delta y} \quad \text{and} \quad z = \frac{z_{\text{physical}} - z_{\min}}{\Delta z}.$$

Then, we consider the integers

$$i_x = floor(x), \quad i_y = floor(y) \quad \text{and} \quad i_z = floor(z), \tag{1}$$

and the normalized offsets (which are real numbers in [0;1))

$$dx = x - i_x, \quad dy = y - i_y \quad \text{and} \quad dz = z - i_z. \tag{2}$$

The cell index i_{cell} is a number in $\{0, 1, \ldots, ncx\cdot ncy\cdot ncz - 1\}$, taken as the image of some one-to-one mapping depending on (i_x, i_y, i_z). For example, the row-major mapping

$$\begin{aligned} &(i_x, i_y, i_z) \mapsto i_{\text{cell}} = (i_x\cdot ncy + i_y)\cdot ncz + i_z \\ &i_{\text{cell}} \mapsto \begin{cases} i_x = \left\lfloor \dfrac{i_{\text{cell}}}{ncz\cdot ncy} \right\rfloor \\ i_y = \text{mod}\left(\left\lfloor \dfrac{i_{\text{cell}}}{ncz} \right\rfloor, ncy \right) \\ i_z = \text{mod}(i_{\text{cell}}, ncz) \end{cases} \end{aligned} \tag{3}$$

is commonly used in `C`. However, several bijection mappings will be analyzed in the following sections with the aim of improving the cache reuse.

Finally, a particle is stored in memory with 1 `int` (i_{cell}), 3 `floats` (dx, dy and dz) and 3 `doubles` (vx, vy and vz). In our code, these 7 attributes are stored in a Structure of Arrays (SoA), to enable efficient vectorization. In this framework, the update particle position step (line 10 in Fig. 1) can be accomplished in the following four sub-steps:

Update positions

1 Compute (i_x, i_y, i_z) from i_{cell}
2 Update (x, y, z) using formula (2) and line 10 in Fig. 1.
3 Compute the new values of $(i_x, dx, i_y, dy, i_z, dz)$ using formulas (1) and (2).
4 Compute i_{cell} from (i_x, i_y, i_z).

The reason behind this approach stems from the fact that on modern architectures, it might be faster to make rapid computations than to store numbers. In this case, we clearly need fast algorithms for computing the bijection functions in sub-steps 1 and 4 above. For example, the bijection mapping in (3) can be computed very fast in both directions, which is not the case for all the mappings analyzed in this paper. In Section 3.2 we show for some bijection mappings that computing them is faster than storing the integers (i_x, i_y, i_z) in addition to i_{cell}.