Cross-platform implementation of Particle-In-Cell method for simulation of high-temperature and fusion plasma by means of hybrid supercomputers equipped with GPU or Intel Xeon Phi accelerators

Alexey A Romanenko¹ and Alexey V Snytnikov²,³ and Marina A Boronina²
¹ Novosibirsk State University
² Institute of Computational Mathematics and Mathematical Geophysics SB RAS
³ Institute of Automation and Electrometry SB RAS
E-mail: arom@nsu.ru, snytav@gmail.com, boronina@ssd.sscc.ru

Abstract. A new Python-based Particle-In-Cell code is presented. The code uses leapfrog particle pusher. The important feature of our code is that all the particles are pushed at once, thus the code is vectorized to improve performance. Electric field is given by Poisson equation with Least squares solver. The code involves collision simulation by PIC-MC method. Both large-scale (MPI) and fine-grain parallelization are being used. The implementation is based on the efficient NumPy library in Python language with the help of Dask package to improve Numpy performance. GPU implementation involves PyCUDA and the performance with Intel Xeon processors and Intel Xeon Phi accelerators is supported by high-performance Intel Python.

1. Introduction
The main idea of this work is to make a very portable implementation of Particle-In-Cell method. This is why Python language was selected as one of the most modern languages for numerical computing which is present almost everywhere and it also has lots of libraries and optimized implementations for various architectures. The second point of this paper is to find out how fast is Python for the real computational physics problems. Another question is whether the high level parallelization tools provide good performance or not?

It should be noticed that Python implementation of PIC method may result performance low compared to the best optimized codes implemented in C++ or Fortran. Then what is the purpose of this work? Why making a code which is a priori slow? The answer is the following: first, it is portable and second, there are a lot of high performance tools implemented for Python which enable further speedup of the code.

The physical problems in scope are the fusion plasma in different experimental facilities and also in astrophysical applications like supernovae expansions and solar bursts of II and III type. Although fusion plasma is generally treated as being collision-less, there are cases when collisions might play a role. Moreover, we intend to use the same Particle-In-Cell engine for low temperature plasma problems, too. An important example of such a problem is the Plasma Enhanced Chemical Vapour Deposition (PECVD) reactor.
Like other plasma physics problems with non-Maxwellian velocity distribution this problem requires kinetic treatment. It means either direct finite-difference solution of Vlasov equation or Particle-In-Cell method [1]. While PIC method is noisy and time-consuming, still it is in many cases the only method that is able to treat the problem with no non-physical simplifications.

There are a lot of PIC packages, like OSIRIS [2], LCODE [3] or VLPL [4], UMKA [5], including those adopted for GPU, e.g. PIConGPU [6], ALaDyn [7], PICADOR [8], and also some works aimed to particular questions of PIC method [9] or of GPU implementation of PIC method, like [10] and [11], but since they are all problem-specific, we make an implementation on our own.

PIC method consists of two stages: field evaluation and particle pushing. Computational experiments with general purpose computers (e.g. the Lomonosov supercomputer at MSU) show that for the collision-less case most time is spent in particle pushing (from 60% to 90% according to [12]). Since the total time is very big even for a small test case: about a week with 128 Intel Xeon cores for a 3D problem with just $100 \times 4 \times 4$ mesh nodes and 10000 particles for each cell [13], it requires using GPUs to solve real-size problem.

2. Basic equations
The basic equations are Vlasov equations for ion and electron components of the plasma and also of the Maxwell equation system. These equations in the usual notation have the following form:

$$\frac{\partial f_{i,e}}{\partial t} + \vec{v} \frac{\partial f_{i,e}}{\partial \vec{r}} + \vec{F}_{i,e} \frac{\partial f_{i,e}}{\partial \vec{p}} = St\{f\}, \quad \vec{F}_{i,e} = q_{i,e} \vec{E}$$

$$\nabla^2 \varphi = 4\pi \rho$$

(1)

Here $f_{i,e}$ is the distribution function for ions and electrons, respectively, $t$ is time, $\vec{v}$ is the velocity vector, $\vec{r}$ is the coordinate vector, $\vec{F}$ is the Lorentz force, $\vec{p}$ is the impulse, $\vec{E}$ is the electric field, $\varphi$ is the electric potential, $\rho$ is the charge density. Finally, $St\{f\}$ is the collision term.

We use the Poisson solver to evaluate electric field.

3. Computational domain
The 3D computational domain has the shape of a cube with the following dimensions:

$$0 \leq x \leq L_X, \quad 0 \leq y \leq L_Y, \quad 0 \leq z \leq L_Z$$

Within this domain there is the model plasma that consists from electrons and ions. The model plasma particles are distributed uniformly within the domain. The density of plasma is set by the user as well as the electron temperature. The temperature of ions is considered to be zero. Initial distribution of particles by velocities is Maxwellian:

$$f(v) = \frac{1}{\Delta v \sqrt{2\pi}} \exp \left( -\frac{(v - v_0)^2}{2\Delta v^2} \right)$$

here $\Delta v$ - electron velocity dispersion, $v_0$ - average velocity ($v_0 = 0$ for plasma particles. Up to this moment it is the same for fusion plasma and for PECVD reactor plasma. But there are also differences. It is about the composition of plasma and the boundary conditions. In this case, the boundary conditions are periodical, and plasma consists of three sorts of particles:

- hydrogen ions,
- plasma electrons,
• beam electrons.

Beam electrons are also distributed uniformly, but they have the same impulse directed along X axis and low temperature along Y and Z.

4. Python implementation

Particle-In-Cell method implementation was inspired by the book [14].

4.1. Particle pusher

Particle pusher is shown in the listing. It uses leapfrog scheme. One can see that the pusher contains no loops, so it is fully vectorized and thus utilizes all the power of Numpy arrays. Some lines of the code, like field evaluation, involve 3D matrices (tensors), so probably faster implementation might be obtained with e.g. TensorFlow.

```python
import numpy as np

def push(x, v, x0, dh, ef, charge, mass, dt):
    lc = np.divide(np.subtract(x, x0), dh)  # cells’ numbers
    lc_int = lc.astype(int)  # integer part of the cells’ numbers
    # float part of the cells’ numbers (or particle weights).
    # at this point, three numbers for each particle, a 3xN array
    weights = np.subtract(lc, lc_int)

    # vector of node weights for all particles
    weights_all = [get_all_weights(l) for l in weights]
    cell_numbers_all = [get_all_nodes(l) for l in lc]
    field_in_points = [[ef[l] for l in group] for group in cell_numbers_all]
    field_in_points = np.array(field_in_points)

    # electric field acting on each particle
    efs = [np.matmul(w, f) for (w, f) in zip(weights_all, field_in_points)]
    ef_part = np.array(efs)* (dt * charge / mass)
    v1 = np.add(v, ef_part)
    v_dt = v * dt
    x1 = np.add(x, v_dt)

    return [x1, v1]
```

4.2. Poisson solver

There are a lot of Poisson equation solvers, but since we have a sparse 7-diagonal matrix, we choose one of the available sparse solvers, namely least squares solver (LSQR).

```python
import cupyx
import cupy as cp

# Copy rhs to GPU
db = cp.asarray(b)
# Assemble sparse matrix
da_csr = cupyx.scipy.sparse.diags(diagonals, offsets, format='csr')
# find the solution
dx = cupyx.scipy.sparse.linalg.lsqr(da_csr, db)
```
Table 1. The Poisson code worktime (in seconds) with different domain sizes

|        | 10^3 | 20^3 | 25^3 |
|--------|------|------|------|
| SciPy dense | 0.03 | 8.56 | 59.70 |
| SciPy sparse | 0.02 | 0.14 | 0.71  |
| CuPy dense | 0.61 | 0.80 | 1.88  |
| CuPy sparse | 0.03 | 0.74 | 1.99  |

Table 2. Results of profiling the pusher part with NVVP

| Total time | # Calls | API call   |
|------------|--------|------------|
| 54.25%     | 5.31818s | cuLaunchKernel |
| 22.25%     | 2.18061s | cudaMalloc  |
| 7.02%      | 687.88ms | cudaGetDevice |

5. Implementation of the PIC method on GPU

There are several approaches how one can move calculations from Python to GPU. One can use PyCUDA [15] and write CUDA kernels directly in the Python code. Another approach is to use CuPy [16] and substitute NumPy arrays with an implementation of NumPy-compatible multi-dimensional array on CUDA. This is the most trivial approach in terms of required manpower resources. The third approach is to implement CUDA functions as a standalone library and access it with Python’s wrappers. The last approach should give the same performance results as the first on and will require the same time for development and debugging. We investigated using CuPy library for this article.

Poisson equation is solved by the least-squares (LSQR) method from the SciPy [17] linear algebra package. To solve the Poisson equation from 1 we convert it to a system of linear equations \(-Ax = b\). The matrix \(A\) has only 7 diagonals with non-zero elements. SciPy provides several routines to solve such systems. We tested routines which works with dense and sparse matrices. The results are presented in the table 1. All codes were tested on Intel E5-2698 v4 and NVIDIA V100 GPU.

A similar approach was used to move the pusher to GPU. Unfortunately, we got performance regression on GPU compare to the CPU implementation. The NVIDIA profiler shows that hundreds of thousands kernels were launched, hundreds of thousands memory allocations were made for one call of the pusher (see table 2).

In such a way, using CuPy library is not suitable here. We plan to use PyCUDA to move the code to GPU.

6. Conclusion

We have implemented Particle-In-Cell method in Python in a fully vectorized form. Thus, we show that Python could be also used as a tool in 3D plasma simulation. Vectorized implementation enables further high-performance implementation of PIC method with BLAS or other low-level tools. Our code is very portable because of Python and it is also high-level code. We need to say that the code was developed and debugged using rather small meshes. For the real case one need to have much bigger mesh size and number of particles. Our code can be parameterized and can be easily adapted on different parallel architectures.
Acknowledgments
The publication has been prepared with the support of RFBR grant 18-07-00364, code development was supported by RFBR grant 19-07-00085. High performance simulations were conducted under state contract with ICMMG SB RAS (0315-2019-0001).

References
[1] Vshivkov V A, Grigoryev Y N and Fedoruk M P 2002 Numerical Particle-in-Cell Methods. Theory and applications (Utrecht-Boston: VSP)
[2] Fonseca R A, Silva L O, Tsung F S, Decyk V K, Lu W, Ren C, Mori W B, Deng S, Lee S, Katsouleas T and Adam J C 2002 Computational Science — ICCS 2002 ed Sloom P M A, Hoekstra A G, Tan C J K and Dongarra J J (Berlin, Heidelberg: Springer Berlin Heidelberg) pp 342–351 ISBN 978-3-540-47789-1
[3] Lotov 2012 (accessed August 11, 2018) LCODE main site URL http://www.inp.nsk.su/~lotov/lcode/
[4] Tuckmantel T, Pukhov A, Liljo J and Hochbruck M 2010 IEEE Transactions on Plasma Science 38 2383–2389
[5] Vshivkov V A, Naumova N M, Pegoraro F and Bulanov S V 1998 Physics of Plasmas 5 2727–2741 (Preprint https://doi.org/10.1063/1.872961) URL https://doi.org/10.1063/1.872961
[6] Burau H, Widera R, Honig W, Juckeland G, Debus A, Kluge T, Schramm U, Cowan T E, Sauerbrey R and Buss M 2010 IEEE Transactions on Plasma Science 38 2831–2839
[7] Rossi F, Londrillo P, Sgattoni A, Sinigardi S and Turchetti G 2012 AIP Conference Proceedings 1507(1) 184–192 (Preprint https://doi.org/10.1063/1.4773692) URL https://doi.org/10.1063/1.4773692
[8] Muravev A, Bashinov A, Elemenko E, Volokitin V, Meyerov I and Gonoskov A 2020
[9] A Yu Perepelkina V D Levchenco I A G 2013 Matem. Mod. (11) 98–110
[10] Kong X, C Huang M, Ren C and K Decyk V 2011 230 1676–1685
[11] Rieke M, Trost T and Grauer R 2015 Journal of Computational Physics 283 436–452 (Preprint https://doi.org/10.1016/j.jcp.2014.12.016) URL https://doi.org/10.1016/j.jcp.2014.12.016
[12] Snytnikov A 2010 Procedia Computer Science 1 607 – 615 ISSN 1877-0509 iCCS 2010 URL http://www.sciencedirect.com/science/article/pii/S1877050910000669
[13] Lotov K V, Timofeev I V, Mesyats E, Snytnikov A V and Vshivkov V A 2015 Physics of Plasmas 22(2) (Preprint https://doi.org/10.1063/1.4907223) URL https://doi.org/10.1063/1.4907223
[14] Brieda L 2019 Plasma Simulation by example (CRC Press)
[15] Pycuda’s documentation URL https://documen.tician.de/pycuda/
[16] Cupy : Numpy-like api accelerated with cuda URL https://github.com/cupy/cupy
[17] Scipy documentation URL https://docs.scipy.org/doc/scipy/reference/