Sequential optimization and parallelization of a Particle-In-Cell code

Yann Barsamian, Sever Hirstoaga, Eric Violard

Inria, TONUS and **IRMA, Université de Strasbourg**

NUMKIN 2016
Strasbourg, October 18, 2016

Outline

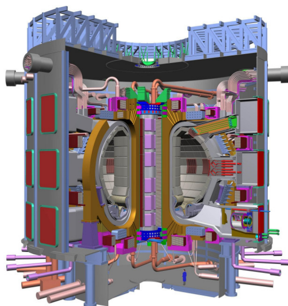
1. Framework
2. Sequential code
3. Parallelization

Outline

1. Framework
2. Sequential code
3. Parallelization

General context

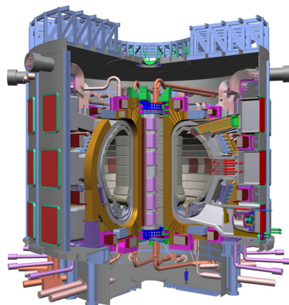
- ITER project (nuclear fusion reactions ⇒ energy)
- plasma magnetic confinement in a tokamak
- strong magnetic field



- Kinetic modeling for electrons by Vlasov-Poisson equations
- 2d2v Vlasov-Poisson with strong magnetic field
- Multiscale behaviour of the solutions
- Large size numerical problems

General context

- ITER project (nuclear fusion reactions $\Rightarrow$ energy)
- plasma magnetic confinement in a tokamak
- strong magnetic field



- Kinetic modeling for electrons by Vlasov-Poisson equations
- 2d2v Vlasov-Poisson with strong magnetic field
- Multiscale behaviour of the solutions
- Large size numerical problems

Numerical approach

Particle-In-Cell method:
The unknown is approximated by a collection of macroparticles $\left(\mathbf{X}_k(t), \mathbf{V}_k(t)\right)_k$

$$f_{N_p}(t, \mathbf{x}, \mathbf{v}) = \sum_{k=1}^{N_p} \omega_k \, \delta(\mathbf{x} - \mathbf{X}_k(t)) \, \delta(\mathbf{y} - \mathbf{V}_k(t))$$

which move along the characteristic curves of Vlasov equation

$$\begin{cases} \mathbf{X}_k'(t) = \mathbf{V}_k(t), \\ \mathbf{V}_k'(t) = \mathbf{V}_k(t) \times \mathbf{B} + \mathbf{E}(t, \mathbf{X}_k(t)) \end{cases} \quad + \quad \mathrm{I.\,C.}$$

AIM: Compute **efficiently** reference solutions for the 4d Vlasov-Poisson systems (multiscale behaviour and long time dynamics) with **classical numerical schemes**.

Particle-In-Cell algorithm

Initialize particles: x, v.

One time step

1. For each particle:
   - interpolate $\mathbf{E}$ in each particle.
   - update v.
   - update x.
   - deposit the charge on the nearest cells.
2. Solve Poisson equation.

Difficulties

- we address noise by using a large number of particles.
- conventional PIC suffers from frequent data movement (memory/CPU).

Particle-In-Cell algorithm

Initialize particles: x, v.
One time step

1. For each particle:
    - interpolate $\mathbf{E}$ in each particle.
    - update v.
    - update x.
    - deposit the charge on the nearest cells.
2. Solve Poisson equation.

Difficulties

- we address noise by using a large number of particles.
- conventional PIC suffers from frequent data movement (memory/CPU).

Particle-In-Cell algorithm

Initialize particles: x, v.

One time step

1. For each particle:
   - interpolate $\mathbf{E}$ in each particle. $\rightsquigarrow$ communications particles $\longleftrightarrow$ field
   - update v. $\rightsquigarrow$ computations
   - update x.
   - deposit the charge on the nearest cells.
2. Solve Poisson equation.

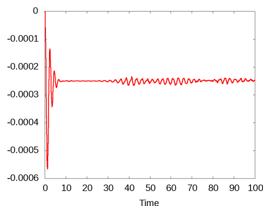
Difficulties

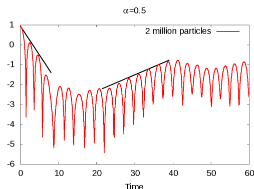
- we address noise by using a large number of particles.
- conventional PIC suffers from frequent data movement (memory/CPU).

Implementation framework

– in SeLaLib: `http://selalib.gforge.inria.fr/`

- Random initial particles
- Linear and cubic splines
- time stepping: leap-frog, RK2
- Poisson equation: Cartesian grid, periodic boundary condition, FFT
- **Landau damping**: verification (conservation of the total energy, electric field energy)





- **guiding center model**, **4d VP with strong magnetic field**.

Verify the code on a convergence result: Long time (~ 1) simulations of

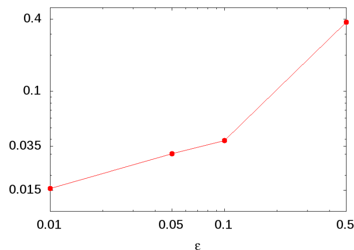
$$\begin{cases} \partial_t g^\varepsilon + \frac{1}{\varepsilon}\Big(\mathbf{v}\cdot\nabla_\mathbf{x} g^\varepsilon + \Big(\mathcal{E}^\varepsilon + \frac{1}{\varepsilon}\mathbf{v}^\perp\Big)\cdot\nabla_\mathbf{v} g^\varepsilon\Big) = 0 \\ + \text{ Poisson} \end{cases}$$

and of the limit model

$$\partial_t g_{GC} + \mathbf{E}^\perp \cdot \nabla_\mathbf{x} g_{GC} = 0 \quad + \text{ Poisson.}$$

- Kelvin-Helmholtz test-case
- $f_0(\mathbf{x},\mathbf{v}) = \frac{1}{2\pi}\Big(\sin(x_2) + 0.05\cos(x_1/2)\Big)\exp\Big(-\frac{v_1^2+v_2^2}{2}\Big)$
- 10 million particles, 256×128 cells

Global relative error at $t = 5$: $||g_{GC} - \rho^\varepsilon||_{L^2} / ||g_{GC}||_{L^2}$



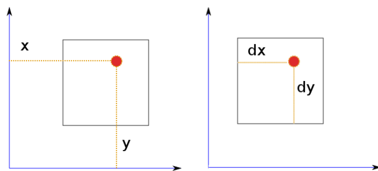
Previous work

E. Chacon-Golcher, SH, M. Lutz: ESAIM Procs, 2016

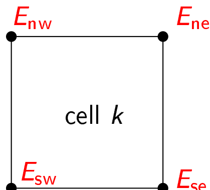
Aims:

- optimize the serial implementation: K. Bowers: 2001, 2008; `VPIC code`
- parallelization (multiprocess & multithreading)

Main ingredients: array of structures (AoS), redundant cell-based E/ρ




```
type :: sll_particle_2d
    sll_int32  :: ic
    sll_real32 :: dx
    sll_real32 :: dy
    sll_real64 :: vx
    sll_real64 :: vy
    sll_real32 :: q
end type sll_particle_2d
```




```
type charge_accumulator_cell_2d
    sll_real64 :: q_sw
    sll_real64 :: q_se
    sll_real64 :: q_nw
    sll_real64 :: q_ne
end type charge_accumulator_cell_2d
```

Previous work

Additional ingredients

- **Particle sorting** (to be done periodically)
- **Hybrid parallelization** MPI/OpenMP
  - distributed memory: **1 process** ⇝ **a list** of particles over the **whole domain**
  - shared memory: assign different segments of the list to different threads.

Performance on one node of the whole performance (1000 iterations, grid with 512x16 cells):

Simulation\Cores	1	2	4	8	16
2 million particles	28.7	56.6	104.7	175.4	252.0
20 million particles	28.6	57.4	106.5	185.4	288.2
200 million particles	28.0	56.9	103.5	174.4	263.8

Number of particles processed per second (in millions): $N_p * N_{iter} / T$.

Previous work

Additional ingredients

- **Particle sorting** (to be done periodically)
- **Hybrid parallelization** MPI/OpenMP
  - distributed memory: **1 process** $\rightsquigarrow$ **a list** of particles over the **whole domain**
  - shared memory: assign different segments of the list to different threads.

Performance on one node of the whole performance (1000 iterations, grid with 512x16 cells):

Simulation\Cores	1	2	4	8	16
2 million particles	28.7	56.6	104.7	175.4	252.0
20 million particles	28.6	57.4	106.5	185.4	288.2
200 million particles	28.0	56.9	103.5	174.4	263.8

Number of particles processed per second (in millions): $N_p * N_{iter} / T$.

Outline

1 Framework

2 Sequential code

3 Parallelization

Current development

Aim: take advantage of SIMD architecture.

Enable vector performances (with both Intel and Gnu compilers) for deposit the charge & update x loops.

Need to comply with[1] cache reuse, memory alignement, unit-stride accessed data

The new optimizations

- Particles in Structure of Arrays (SoA).
- Redundant cell-based structure for $\mathbf{E}/\rho$, coupled with space-filling curves for decrease of the cache misses.
- loop transformation & code rewriting for automatic vectorization of particles deposit and update-positions loops.

[1] H. Vincenti et al., "An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes", 2016, CPC.

Current development

Aim: take advantage of SIMD architecture.

Enable vector performances (with both Intel and Gnu compilers) for deposit the charge & update x loops.

Need to comply with[1] cache reuse, memory alignement, unit-stride accessed data

The new optimizations

- Particles in Structure of Arrays (SoA).
- Redundant cell-based structure for $\mathbf{E}/\rho$, coupled with space-filling curves for decrease of the cache misses.
- loop transformation & code rewriting for automatic vectorization of particles deposit and update-positions loops.

[1] H. Vincenti et al., "An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes", 2016, CPC.

Loop transformation

1. For each particle:
   - interpolate **E** in each particle.
   - update v.
2. For each particle:
   - update x.
3. For each particle:
   - deposit the charge on the nearest cells.
4. Solve Poisson equation.

Reasons:

- efficiently vectorize each loop **alone**.
- gain of 18%-25% for different data structure.

Rewriting the deposit

Standard deposit: Matrix of ncx rows and ncy columns

```
double rho[ncx][ncy];
```

rho[i_x][i_y] += w*(1-dx[i])*(1-dy[i]);
rho[i_x][i_y+1] += w*(dx[i])*(1-dy[i]);
rho[i_x+1][i_y] += w*(1-dx[i])*(dy[i]);
rho[i_x+1][i_y+1] += w*(dx[i])*(dy[i]);

Redundant cell-based deposit[2]:
Array of ncx*ncy cells: each cell contains data for the four corners.

```
double rho[ncx*ncy][4];
```

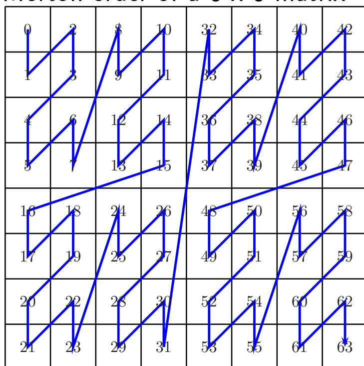
for (corner = 0; corner < 4; corner++)
rho[i_{cell}[i]][corner] += w * (cx[corner] + sx[corner] * dx[i])
* (cy[corner] + sy[corner] * dy[i]);

[2] H. Vincenti et al., "An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes", 2016, CPC.

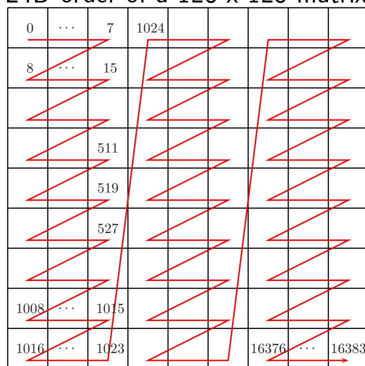
Data structure for $\mathbf{E}/\rho$

Space-filling curves

Morton-order of a 8 x 8 matrix

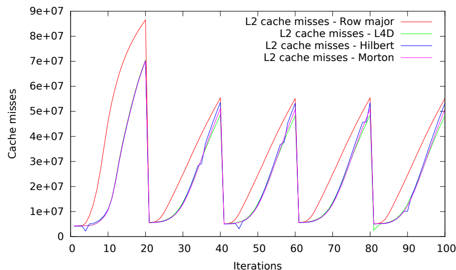


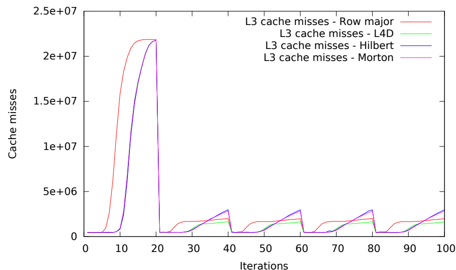
L4D-order of a 128 x 128 matrix



Efficiency comparison

Number of cache misses[3] of different curves: Morton, L4D, Hilbert, and clasicall row-major.





[3] computed with library PAPI

Overall performance results

- 128 x 128 cells, 50 million particles, 100 iterations (sort every 20)
- Structure of Arrays (automatic vectorization of the Update x step)

Time spent in different loops (in seconds):

	Update v	Update x	Deposit	Total
standard 2d	30.6	12.5	20.7	74.3
Row-major	32.3	12.8	14.9	70.5
L4D	29.7	15.9	12.7	68.8
Morton	29.6	15.3	12.7	69.0

Thus, 65 million particles processed/second/core on Intel Haswell.

Overall performance results

- 128 x 128 cells, 50 million particles, 100 iterations (sort every 20)
- Structure of Arrays (automatic vectorization of the Update x step)

Time spent in different loops (in seconds):

	Update v	Update x	Deposit	Total
standard 2d	30.6	12.5	20.7	74.3
Row-major	32.3	12.8	14.9	70.5
L4D	29.7	15.9	12.7	68.8
Morton	29.6	15.3	12.7	69.0

Thus, 65 million particles processed/second/core on Intel Haswell.

Outline

1. Framework
2. Sequential code
3. Parallelization

Context

Large problem size $\rightsquigarrow$ distributed memory parallelization - through (coarse grain) domain decomposition. Good scaling up to a few 10^5 processors.

This is not our approach:

Hybrid parallelization: MPI/OpenMP

- distributed memory: **1 process** $\rightsquigarrow$ **a list** of particles over the **whole domain**
- shared memory: assign different segments of the list to different threads.

Applicability to at most 100*s* of cores[4].

[4] K. Germaschewski et al., "The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing", 2016, JCP, pp 305 - 326.

Results on one socket

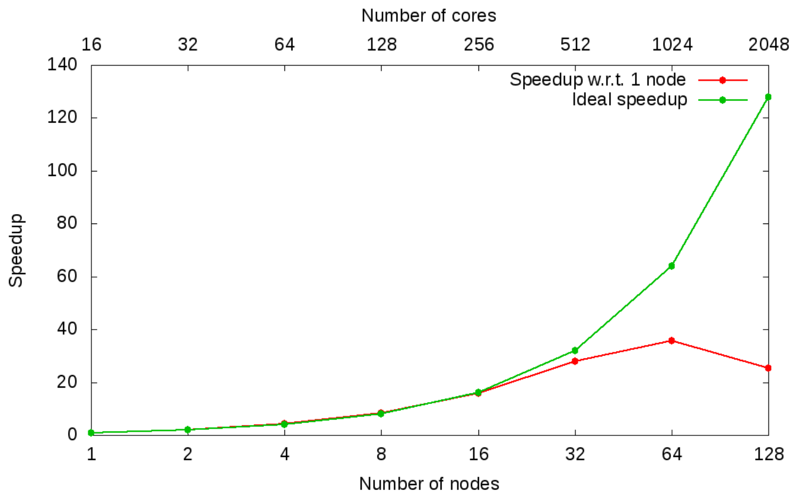
Thanks to Michel Mehrenberger for the DARI 2016 project on GENCI's supercomputer Curie.

One node: 2×8 cores SandyBridge

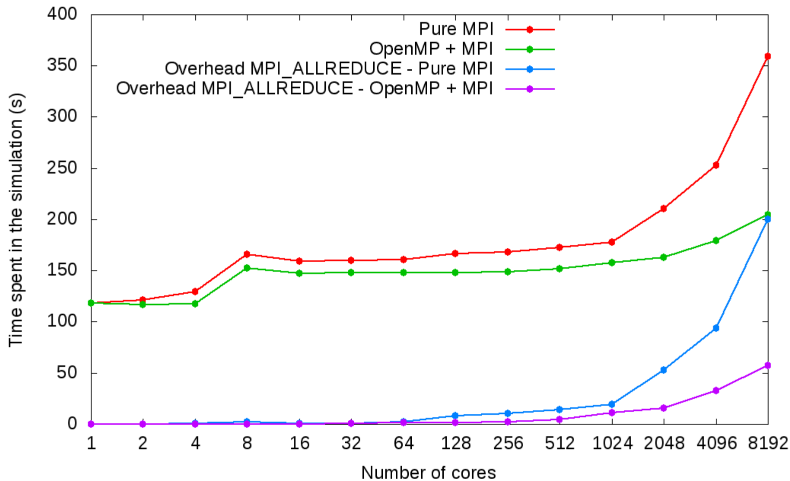
1 core	2 cores	4 cores	8 cores
40.4	79.6	158	249

Number of particles processed per second (in millions) (128 x 128 grid, 50 million particles, 100 iterations simulation, sorting every 20 iterations).

Strong scaling



Weak scaling



Conclusions - Outlook

• Efficient PIC code with classical numerical schemes.

• gain of 36% in cache misses: L4D-ordering vs. row-major.

• SoA is better than AoS for the deposit charge **but not** for the interpolation step
$\star$ to try: use an AoS in memory, change a little portion of the data (before the computations) to benefit from unit stride vectorization.

• a gain of $1.6\times$ on the deposit loop.
$\star$ in $3D$ a better scaling of the deposit loop vectorization (cf. Vincenti 2016)

$\star$ domain decomposition?

THANK YOU!