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 $\Rightarrow$ 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
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
Particle-In-Cell method:
The unknown is approximated by a collection of macroparticles $\left( X_k(t), V_k(t) \right)_k$

$$f_{N_p}(t, x, v) = \sum_{k=1}^{N_p} \omega_k \delta(x - X_k(t)) \delta(y - V_k(t))$$

which move along the characteristic curves of Vlasov equation

$$\begin{cases}
    X_k'(t) = V_k(t), \\
    V_k'(t) = V_k(t) \times B + E(t, X_k(t)) + I. C.
\end{cases}$$

**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 \( E \) 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 $E$ 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 \( E \) in each particle. \( \rightsquigarrow \) communications particles \( \leftrightarrow \) 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

  
  - 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 ($\sim 1$) simulations of

$$\begin{cases}
\partial_t g^\varepsilon + \frac{1}{\varepsilon} \left( \mathbf{v} \cdot \nabla_x g^\varepsilon + \left( \mathcal{E}^\varepsilon + \frac{1}{\varepsilon} \mathbf{v}^\perp \right) \cdot \nabla_v g^\varepsilon \right) = 0 \\
+ \text{Poisson}
\end{cases}$$

and of the limit model

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

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

Global relative error at $t = 5$:

$$\left\| g_{GC} - \rho^\varepsilon \right\|_{L^2} / \left\| g_{GC} \right\|_{L^2}$$
Previous work


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

Main ingredients: array of structures (AoS), redundant cell-based $E/\rho$
Previous work

Additional ingredients

- **Particle sorting** (to be done periodically)
- **Hybrid parallelization** MPI/OpenMP
  - distributed memory: 1 process $\leftrightarrow$ 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):

<table>
<thead>
<tr>
<th>Simulation/Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 million particles</td>
<td>28.7</td>
<td>56.6</td>
<td>104.7</td>
<td>175.4</td>
<td>252.0</td>
</tr>
<tr>
<td>20 million particles</td>
<td>28.6</td>
<td>57.4</td>
<td>106.5</td>
<td>185.4</td>
<td>288.2</td>
</tr>
<tr>
<td>200 million particles</td>
<td>28.0</td>
<td>56.9</td>
<td>103.5</td>
<td>174.4</td>
<td>263.8</td>
</tr>
</tbody>
</table>

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

Yann Barsamian, Sever Hirstoaga, Eric Violard

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

<table>
<thead>
<tr>
<th>Simulation \ Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 million particles</td>
<td>28.7</td>
<td>56.6</td>
<td>104.7</td>
<td>175.4</td>
<td>252.0</td>
</tr>
<tr>
<td>20 million particles</td>
<td>28.6</td>
<td>57.4</td>
<td>106.5</td>
<td>185.4</td>
<td>288.2</td>
</tr>
<tr>
<td>200 million particles</td>
<td>28.0</td>
<td>56.9</td>
<td>103.5</td>
<td>174.4</td>
<td>263.8</td>
</tr>
</tbody>
</table>

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

1 Framework

2 Sequential code

3 Parallelization
**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 alignment, 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 alignment, unit-stride accessed data

The new optimizations

- Particles in *Structure of Arrays (SoA)*.
- Redundant cell-based structure for \(E/\rho\), 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 $E/\rho$

Space-filling curves

<table>
<thead>
<tr>
<th>Morton-order of a 8 x 8 matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L4D-order of a 128 x 128 matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>511</td>
</tr>
<tr>
<td>519</td>
</tr>
<tr>
<td>527</td>
</tr>
<tr>
<td>1008</td>
</tr>
<tr>
<td>1016</td>
</tr>
<tr>
<td>16376</td>
</tr>
</tbody>
</table>
Number of cache misses\textsuperscript{3} of different curves: Morton, L4D, Hilbert, and classical row-major.

\textsuperscript{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):

<table>
<thead>
<tr>
<th></th>
<th>Update v</th>
<th>Update x</th>
<th>Deposit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard 2d</td>
<td>30.6</td>
<td>12.5</td>
<td>20.7</td>
<td>74.3</td>
</tr>
<tr>
<td>Row-major</td>
<td>32.3</td>
<td>12.8</td>
<td>14.9</td>
<td>70.5</td>
</tr>
<tr>
<td>L4D</td>
<td>29.7</td>
<td>15.9</td>
<td>12.7</td>
<td>68.8</td>
</tr>
<tr>
<td>Morton</td>
<td>29.6</td>
<td>15.3</td>
<td>12.7</td>
<td>69.0</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th></th>
<th>Update $v$</th>
<th>Update $x$</th>
<th>Deposit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard 2d</td>
<td>30.6</td>
<td>12.5</td>
<td>20.7</td>
<td>74.3</td>
</tr>
<tr>
<td>Row-major</td>
<td>32.3</td>
<td>12.8</td>
<td>14.9</td>
<td>70.5</td>
</tr>
<tr>
<td>L4D</td>
<td>29.7</td>
<td>15.9</td>
<td>12.7</td>
<td>68.8</td>
</tr>
<tr>
<td>Morton</td>
<td>29.6</td>
<td>15.3</td>
<td>12.7</td>
<td>69.0</td>
</tr>
</tbody>
</table>

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

1. Framework
2. Sequential code
3. Parallelization
Large problem size $\Rightarrow$ 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 $\Rightarrow$ a list of particles over the whole domain
- shared memory: assign different segments of the list to different threads.

Applicability to at most 100s of cores$^4$.

---

Results on one socket

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

One node: $2 \times 8$ cores SandyBridge

<table>
<thead>
<tr>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.4</td>
<td>79.6</td>
<td>158</td>
<td>249</td>
</tr>
</tbody>
</table>

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

800 millions particles - Grid 256x256 - 100 iterations

Number of cores

Number of nodes

Speedup vs. nodes

Ideal speedup

Speedup w.r.t. 1 node
Sequential optimization and parallelization of a Particle-In-Cell code

50 millions particles / core - Grid 128x128 - 100 iterations

Yann Barsamian, Sever Hirstoaga, Eric Violard
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
  ⋆ 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× on the deposit loop.
  ⋆ in 3D a better scaling of the deposit loop vectorization (cf. Vincenti 2016)

⋆ domain decomposition?
THANK YOU!