Optimization of Particle-in-Cell methods : 2d2v Vlasov-Poisson equation, one and two species

Y. Barsamian, J. Bernier, S. Hirstoaga, M. Mehrenberger, É. Violard

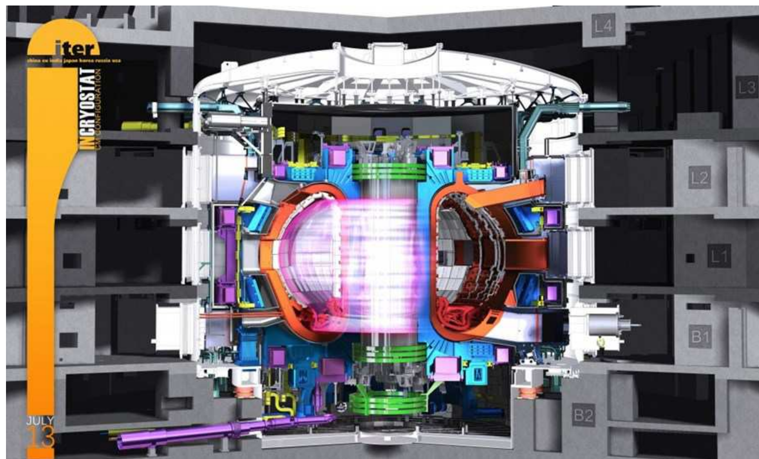
Universities of Rennes & Strasbourg

November 2016

Outline

- Physical & mathematical background.
- Optimization of the sequential code.
- Other results.

General context



ITER[1] tokamak (controlled thermonuclear fusion)

1. « The way » (in Latin) to produce energy

Kinetic Modeling : one species

$$\begin{cases} \dfrac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f - \dfrac{\mathrm{e}}{m} \vec{E} \cdot \nabla_{\vec{v}} f = 0 & \text{Vlasov} \\ -\Delta \phi = \rho & \text{Poisson} \end{cases}$$

- $f(\vec{x}, \vec{v}, t)$: distribution function of the electrons
- $\vec{E}(\vec{x}, t) = -\overrightarrow{grad}\,\phi$: the electric field, here self-induced ; ϕ is the associated scalar potential
- e, m : electron charge and mass
- t : time
- $\vec{x}$: particle position (1d, 2d or 3d), periodic boundaries
- $\vec{v}$: particle velocity (1d, 2d or 3d)
- $\rho(\vec{x}, t) = \mathrm{e}\left(1 - \int f(\vec{x}, \vec{v}, t) d\vec{v}\right)$: volume charge density

Kinetic Modeling : two species

$$\begin{cases} \dfrac{\partial f_i}{\partial t} + \overrightarrow{v} \cdot \nabla_{\overrightarrow{x}} f_i + \mathrm{e}\overrightarrow{E} \cdot \nabla_{\overrightarrow{v}} f_i = 0 & \text{Vlasov}_i \\ \dfrac{\partial f_e}{\partial t} + \dfrac{1}{\varepsilon}\overrightarrow{v} \cdot \nabla_{\overrightarrow{x}} f_e - \dfrac{\mathrm{e}}{\varepsilon}\overrightarrow{E} \cdot \nabla_{\overrightarrow{v}} f_e = 0 & \text{Vlasov}_e \\ -\Delta\phi = \rho & \text{Poisson} \end{cases}$$

- $f_i(\overrightarrow{x}, \overrightarrow{v}, t)$: distribution function of the ions
- $f_e(\overrightarrow{x}, \overrightarrow{v}, t)$: distribution function of the electrons
- $\varepsilon = \sqrt{\dfrac{m_e}{m_i}}$: square root of mass ratio
- $\rho(\overrightarrow{x}, t) = \mathrm{e}\left(\int (f_i(\overrightarrow{x}, \overrightarrow{v}, t) - f_e(\overrightarrow{x}, \overrightarrow{v}, t))d\overrightarrow{v}\right)$: volume charge density

Particle-in-Cell Methods

- discretization of f via (a lot of) numerical particles
  - one numerical particle represents many real-life particles
  - $f(\overrightarrow{x}, \overrightarrow{v}, t) = \sum_{k=1}^{N} w_k \delta(\overrightarrow{x} - \overrightarrow{x_k}) \delta(\overrightarrow{v} - \overrightarrow{v_k})$

Particle-in-Cell Methods

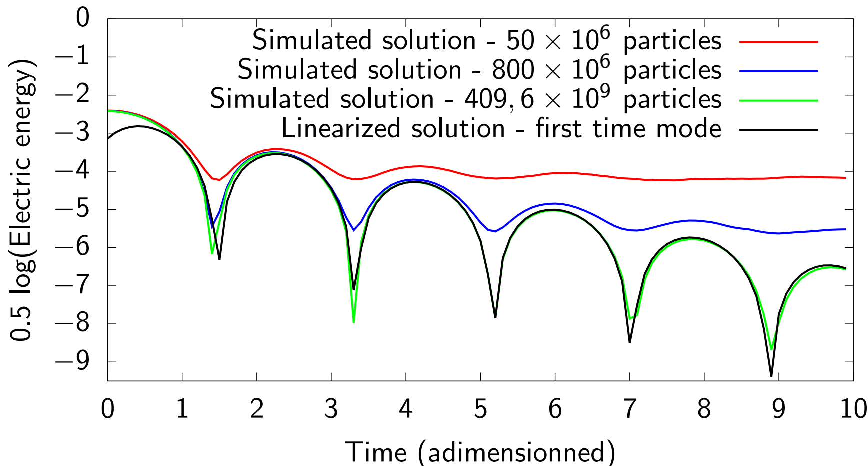
- discretization of f via (a lot of) numerical particles
  - one numerical particle represents many real-life particles
  - $f(\overrightarrow{x}, \overrightarrow{v}, t) = \sum_{k=1}^{N} w_k \delta(\overrightarrow{x} - \overrightarrow{x_k})\delta(\overrightarrow{v} - \overrightarrow{v_k})$
- discretization of ρ (computed from the particles via a deposition step) and $\overrightarrow{E}$ (computed from ρ via a Fourier method) on a 2d grid.

- discretization of f via (a lot of) numerical particles
  - one numerical particle represents many real-life particles
  - $f(\overrightarrow{x}, \overrightarrow{v}, t) = \sum_{k=1}^{N} w_k \delta(\overrightarrow{x} - \overrightarrow{x_k}) \delta(\overrightarrow{v} - \overrightarrow{v_k})$
- discretization of ρ (computed from the particles via a deposition step) and $\overrightarrow{E}$ (computed from ρ via a Fourier method) on a 2d grid.
- interaction of particles only via the self-induced fields : particles move along the characteristic curves of the Vlasov equation

$$\begin{cases} \dfrac{\mathrm{d}\overrightarrow{x}(t)}{\mathrm{d}t} = \overrightarrow{v}(t), \\ \dfrac{\mathrm{d}\overrightarrow{v}(t)}{\mathrm{d}t} = \dfrac{\mathrm{e}}{m} \overrightarrow{E}(\overrightarrow{x}(t), t). \end{cases}$$

Landau Damping, 128×128 grid, $\Delta t = 0.1$

$$\begin{cases} f(\overrightarrow{x}, \overrightarrow{v}, 0) = \dfrac{e^{-\frac{(v_x+v_y)^2}{2}}}{2\pi}(1 + A\cos(k_x x)\cos(k_y y)) \\ \overrightarrow{x} = (x, y) \in \left[0; \frac{2\pi}{k_x}\right) \times \left[0; \frac{2\pi}{k_y}\right), k_x = k_y = 0.5, A = 0.01 \end{cases}$$



PIC Pseudo-Code

Initialization :

1 Initialize *particles* with *num_particles* particles and sort it
2 Compute *rho* and E at $t = 0$

Algorithm :

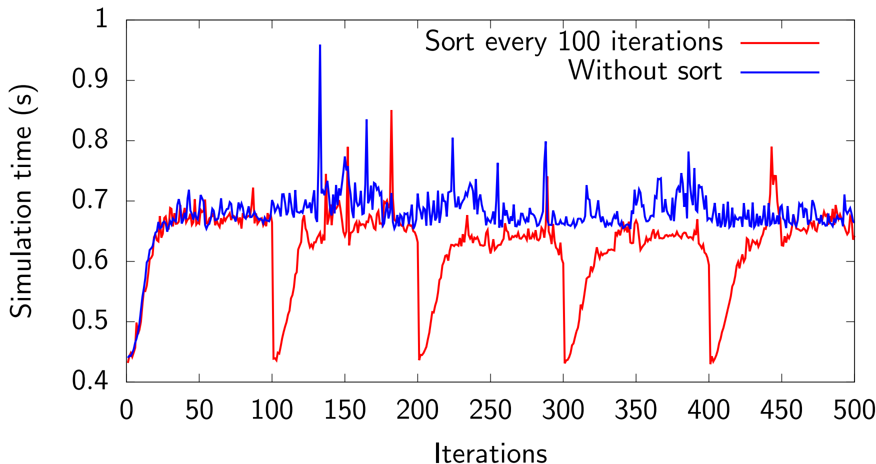
3 **For** *i* from 1 to *num_iterations*, **do**
4   **If** (*condition*), **then**
5   Sort the particles
6   **End If**
7   Set all cells of *rho* to 0
8   **Foreach** particle in *particles*, **do**
9   Update the velocity $v += \frac{q}{m}E$
10   Update the position $x += \Delta_t v$
11   Accumulate the charge on the nearest *rho* cells
12   **End Foreach**
13   Compute E from *rho* Poisson solver
14 **End For**

Previous Work

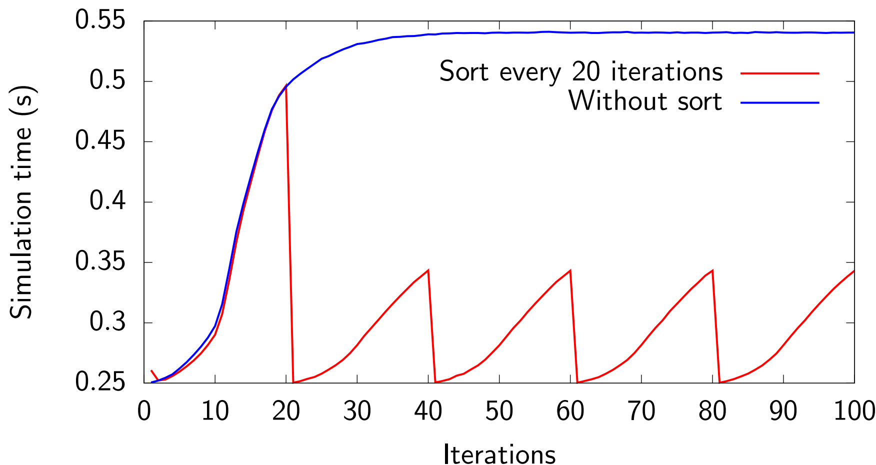
We borrowed many ingredients of our code from :

- Decyk, Karmesin, de Boer & Liewer (1996)
  - sorting of the particle array
- Bowers & Li (2003)
  - data structure for ρ and $\overrightarrow{E}$: redundant, cell-based
- Bowers, Albright, Yin, Bergen & Kwan (2008)
  - data structure for the particles : cell index plus offset
- Chacon-Golcher, Hirstoaga & Lutz (2016)
  - baseline for the code
- Vincenti, Lobet, Lehe, Sasanka & Vay (2016)
  - vectorization of the charge accumulation

Sorting



CPU time for 50 000 000 particles, with a 128×128 mesh, $\Delta t = 0.1$.
Without sort : 588 s ; with sort : 578 s (including 37 s of sorting).
SandyBridge @ 2.7 GHz with 4 GB of RAM (supercomputer Curie)

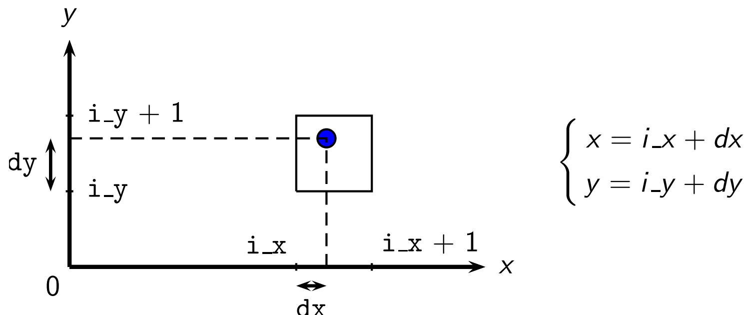


CPU time for 50 000 000 particles, with a 128×128 mesh, $\Delta t = 0.1$.
Without sort : 87 s ; with sort : 67 s (including 8.6 s of sorting).
Haswell @ 2.3 GHz with 32 GB of RAM

Cell Index Plus Offset

Particle at $(x_{\text{physical}}, y_{\text{physical}}) \in [x_{min}; x_{max}) \times [y_{min}; y_{max})$.

Grid discretized among $ncx \times ncy$ points : renormalize $(x, y) \in [0; ncx) \times [0; ncy)$

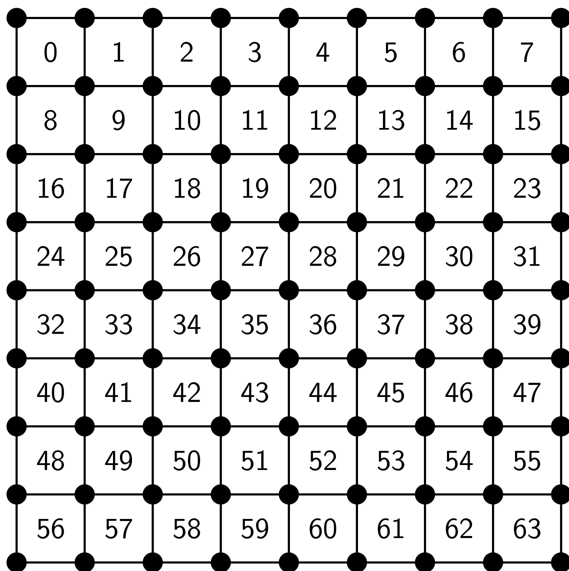


$$\begin{cases} x = i_x + dx \\ y = i_y + dy \end{cases}$$

$i_{cell} \in \{0, 1, \ldots ncx \times ncy - 1\}$: one-to-one correspondence with $(i_x, i_y) \in \{0, 1, \ldots ncx - 1\} \times \{0, 1, \ldots ncy - 1\}$.

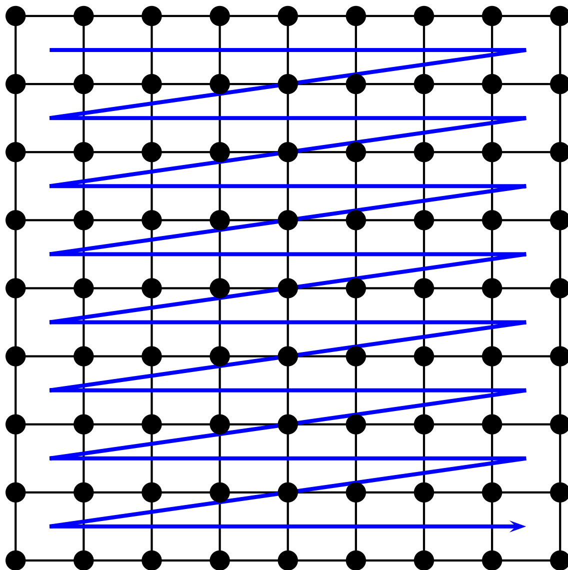
Example : row-major mapping $(i_x, i_y) \mapsto i_{cell} = i_x \times ncy + i_y$.

Redundant Data Structure



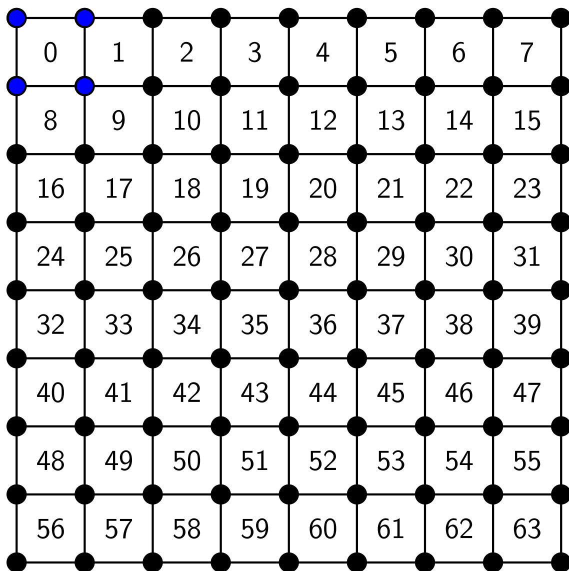
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



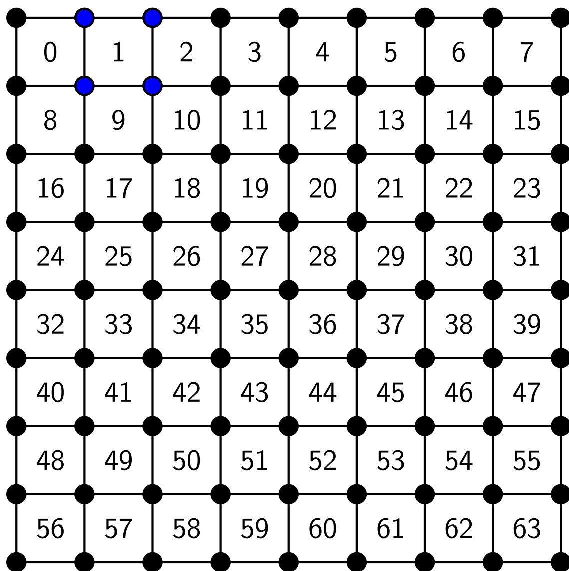
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



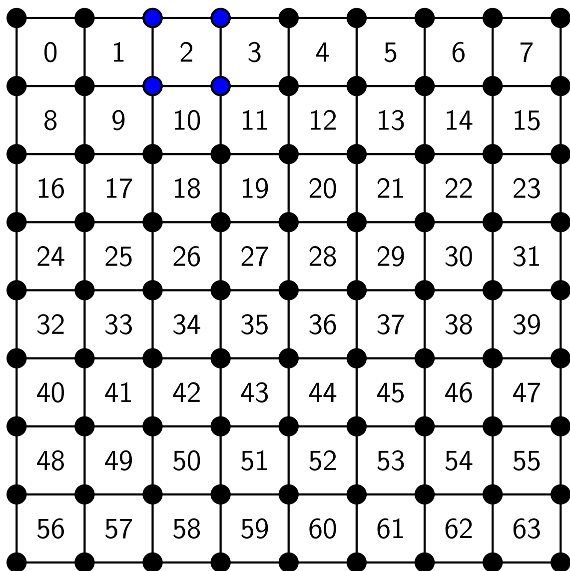
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



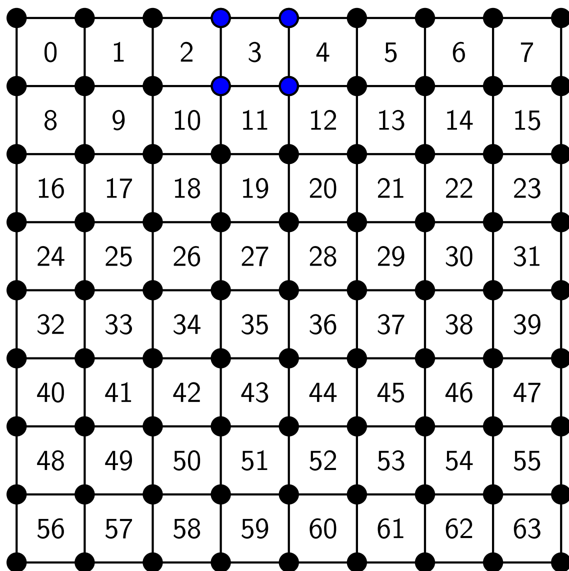
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



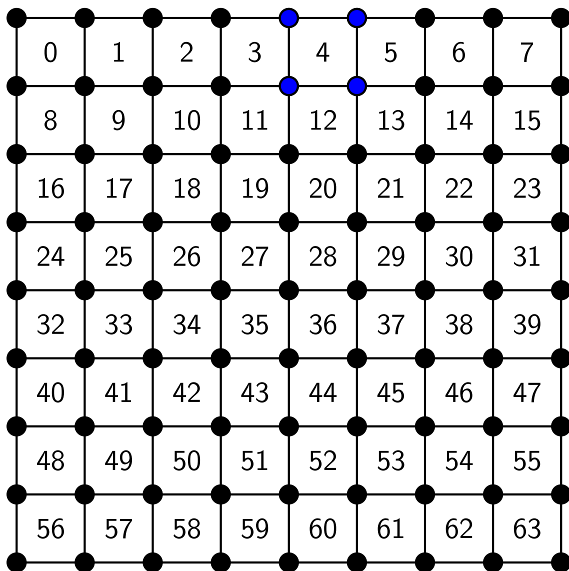
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



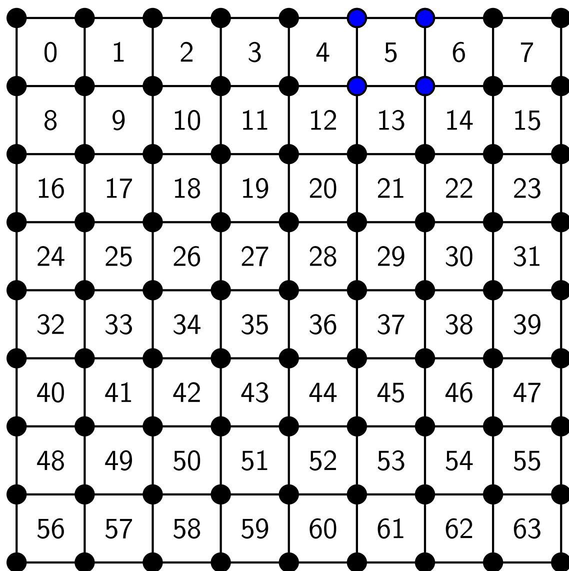
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



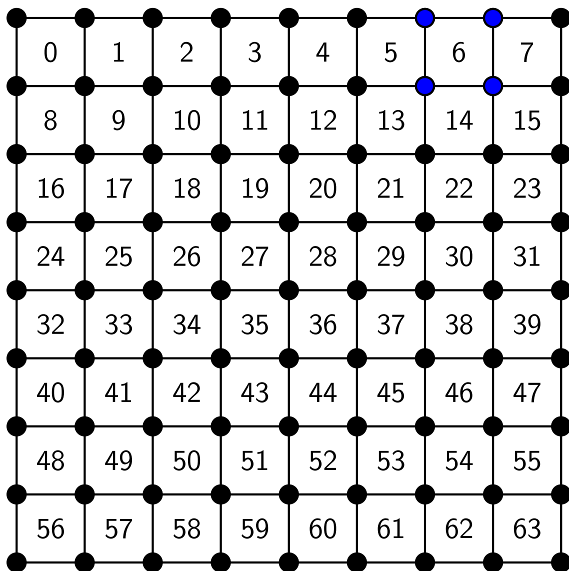
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



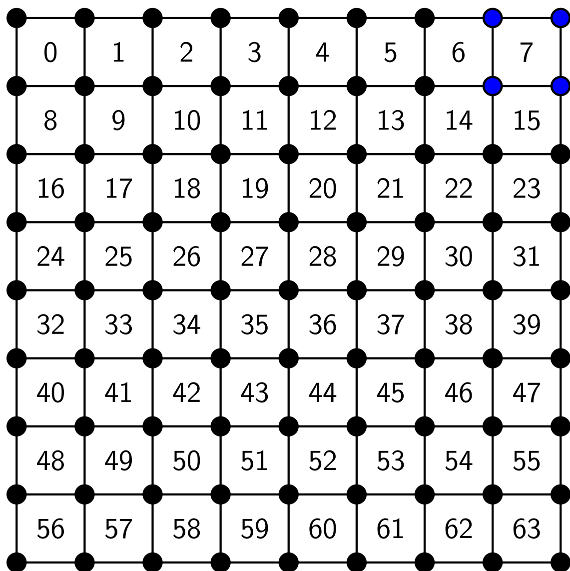
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



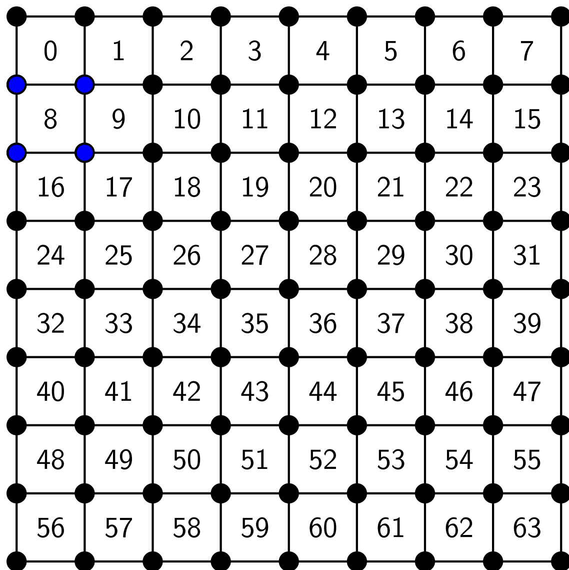
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



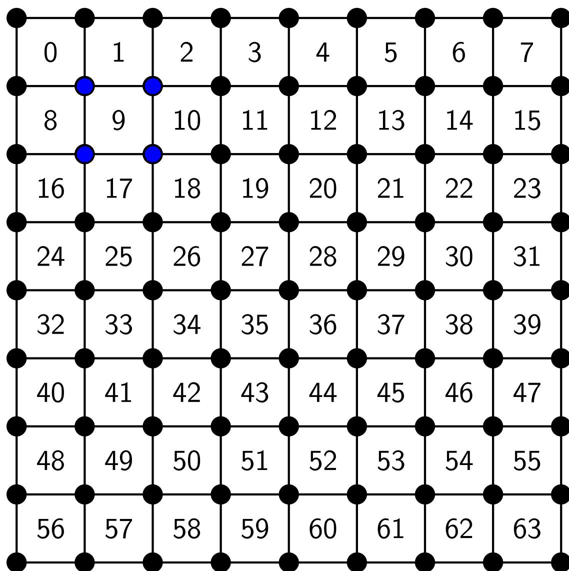
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



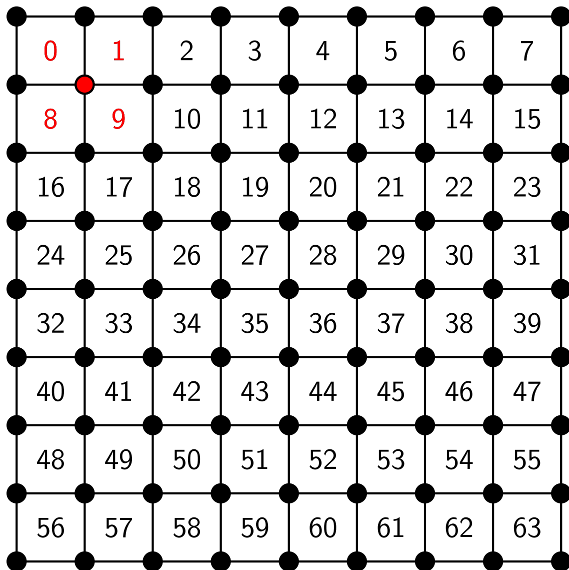
Row-major layout of a 8 x 8 matrix

Redundant Data Structure



Row-major layout of a 8 x 8 matrix

Redundant Data Structure



Row-major layout of a 8 x 8 matrix

Vectorization of the Accumulation (for particle i)

```
// Standard 2d data structure.
double rho[ncx][ncy];
rho[i_x  ][i_y  ] += w * (1 - dx[i]) * (1 - dy[i]);
rho[i_x  ][i_y+1] += w * (1 - dx[i]) * (    dy[i]);
rho[i_x+1][i_y  ] += w * (    dx[i]) * (1 - dy[i]);
rho[i_x+1][i_y+1] += w * (    dx[i]) * (    dy[i]);

// Redundant data structure.
double rho_1d[ncx*ncy][4];
float cx[4] = {  1.,  1.,  0.,  0.};
float sx[4] = { -1., -1.,  1.,  1.};
float cy[4] = {  1.,  0.,  1.,  0.};
float sy[4] = { -1.,  1., -1.,  1.};
for (corner = 0; corner < 4; corner++)
  rho_1d[i_cell[i]][corner] += w *
    (cx[corner] + sx[corner] * dx[i]) *
    (cy[corner] + sy[corner] * dy[i]);
```

Additional Optimizations

Goals :

- good cache reuse (memory accesses are a major bottleneck)
- efficient vectorization of the code (SIMD architectures)

Additional Optimizations

Goals :

- good cache reuse (memory accesses are a major bottleneck)
- efficient vectorization of the code (SIMD architectures)

Means :

- space-filling curves
- structure of arrays
- loop transformations : splitting, hoisting
- removing of `ifs` and function calls (replaced by efficient bitwise operations)

Loop Splitting

```
1    Initialize particles with num_particles particles and sort it
2    Compute rho and E at t = 0
3    For i from 1 to num_iterations, do
4        If (condition), then
5            Sort the particles
6        End If
7        Set all cells of rho to 0
8        Foreach particle in particles, do
9            Update the velocity                                   v+ = (q/m) E
10           Update the position                                   x+ = Δ_t v
11           Accumulate the charge on the nearest rho cells
12       End Foreach
13       Compute E from rho                                        Poisson solver
14   End For
```

Loop Splitting

```
1    Initialize particles with num_particles particles and sort it
2    Compute rho and E at t = 0
3    For i from 1 to num_iterations, do
4        If (condition), then
5            Sort the particles
6        End If
7        Set all cells of rho to 0
8        Foreach particle in particles, do
9            Update the velocity                                   v+ = (q/m) E
         End Foreach
         Foreach particle in particles, do
10           Update the position                                   x+ = Δ_t v
         End Foreach
         Foreach particle in particles, do
11           Accumulate the charge on the nearest rho cells
12       End Foreach
13       Compute E from rho                                        Poisson solver
14   End For
```

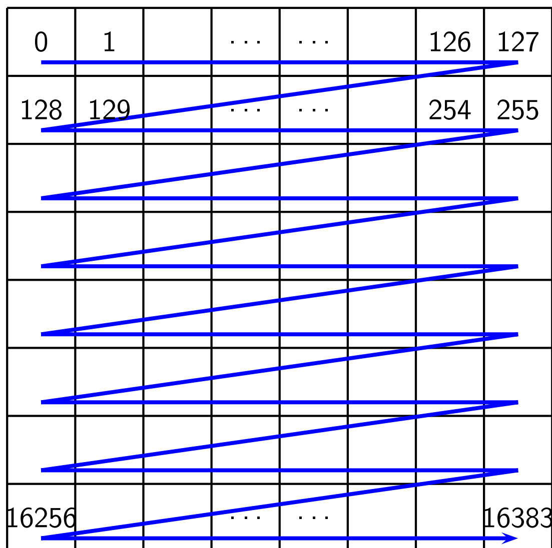
Loop Splitting

```
1     Initialize particles with num_particles particles and sort it
2     Compute rho and E at t = 0
3     For i from 1 to num_iterations, do
4         If (condition), then
5             Sort the particles
6         End If
7         Set all cells of rho to 0
8         Foreach particle in particles, do
9             Update the velocity                                  v+ = (q/m)E
          End Foreach
          Foreach particle in particles, do
10            Update the position                                  x+ = Δt v
          End Foreach
          Foreach particle in particles, do
11            Accumulate the charge on the nearest rho cells
12        End Foreach
13        Compute E from rho                                       Poisson solver
14    End For
```

Loop Splitting

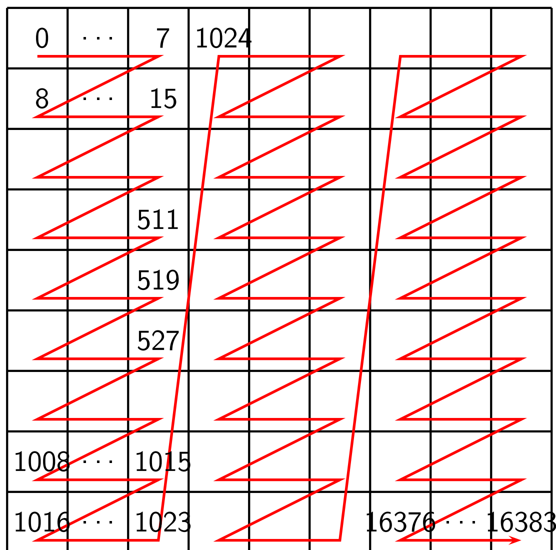
```
1     Initialize particles with num_particles particles and sort it
2     Compute rho and E at t = 0
3     For i from 1 to num_iterations, do
4         If (condition), then
5             Sort the particles
6         End If
7         Set all cells of rho to 0
8         Foreach particle in particles, do
9             Update the velocity ⇝ advection in v⃗
          End Foreach
          Foreach particle in particles, do
10            Update the position ⇝ advection in x⃗
          End Foreach
          Foreach particle in particles, do
11            Accumulate the charge on the nearest rho cells
12        End Foreach
13        Compute E from rho                                Poisson solver
14    End For
```

Space-Filling Curves for E and ρ : Row-Major



Standard layout (in C)

Space-Filling Curves for E and ρ : L4D (SIZE=8)



Chatterjee, Jain, Lebeck, Mundhra, Thottethodi (1999)

Space-Filling Curves for E and ρ : Morton

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

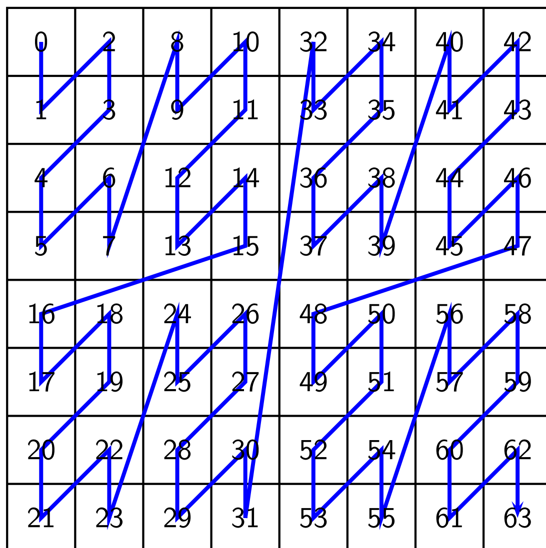
Morton (1966)

Space-Filling Curves for E and ρ : Morton

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

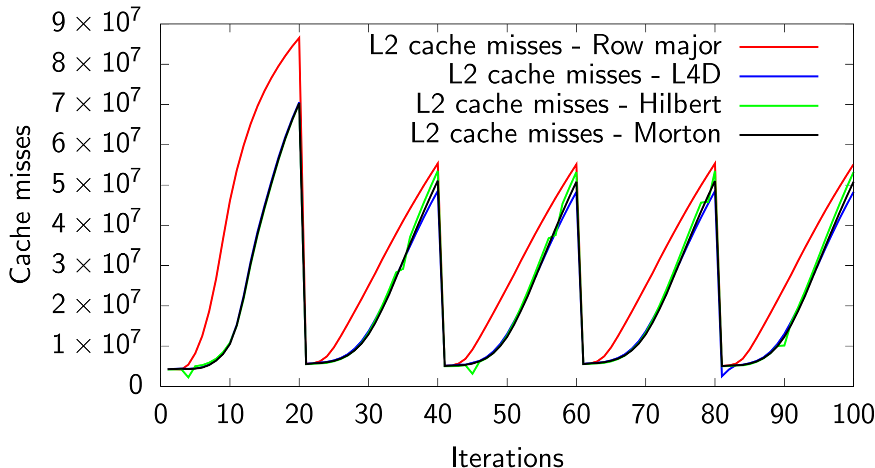
Morton (1966)

Space-Filling Curves for E and ρ : Morton



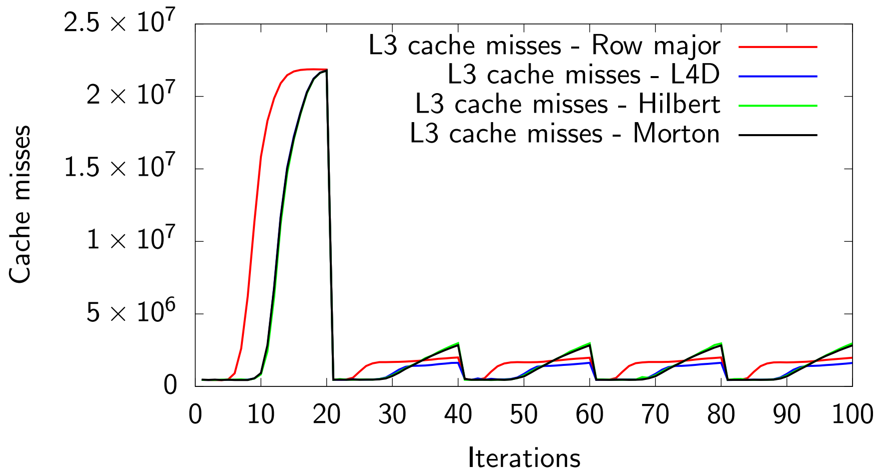
Morton (1966)

Space-Filling Curves : Cache Misses Results



Number of cache misses in the Level 2 Cache for several space-filling curves, for 50 000 000 particles, with a 128×128 mesh, $\Delta t = 0.1$. Haswell @ 2.3 GHz with 32 GB of RAM

Space-Filling Curves : Cache Misses Results



Number of cache misses in the Level 3 Cache for several space-filling curves, for 50 000 000 particles, with a 128×128 mesh, $\Delta t = 0.1$. Haswell @ 2.3 GHz with 32 GB of RAM

Overall Space-Filling Curves Comparisons

	Update v	Update x	Accumulate	Total
2d standard	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
Hilbert	30.0	*133.1*	12.8	185.8

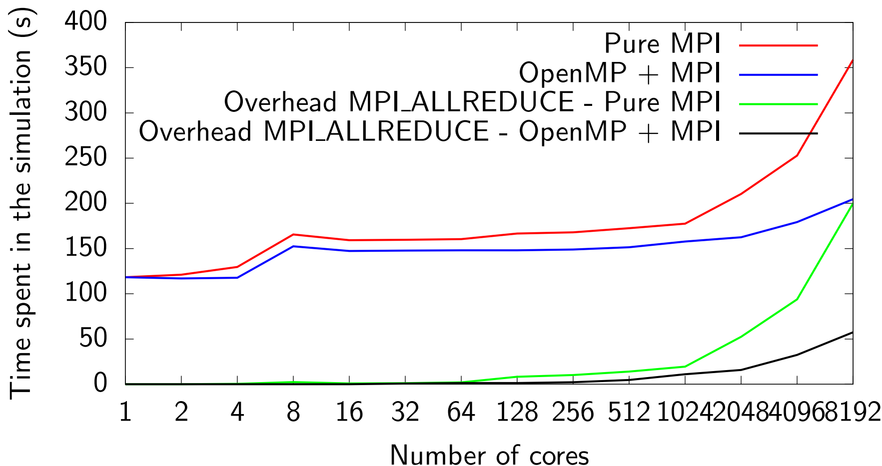
Time spent in the different loops (in seconds), for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 20 iterations).

Overall Space-Filling Curves Comparisons

	Update v	Update x	Accumulate	Total
2d standard	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
Hilbert	30.0	*133.1*	12.8	185.8

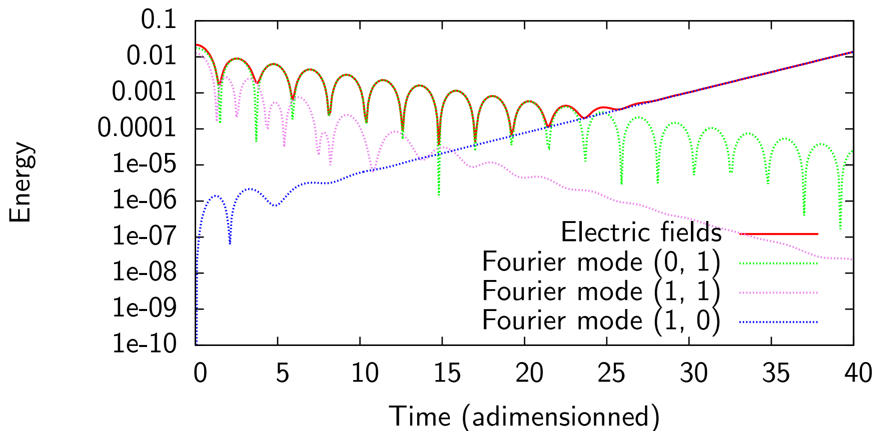
Time spent in the different loops (in seconds), for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 20 iterations).

65 million particles processed/second/core on Intel Haswell.
43 million particles processed/second/core on Intel Sandy Bridge (Curie).



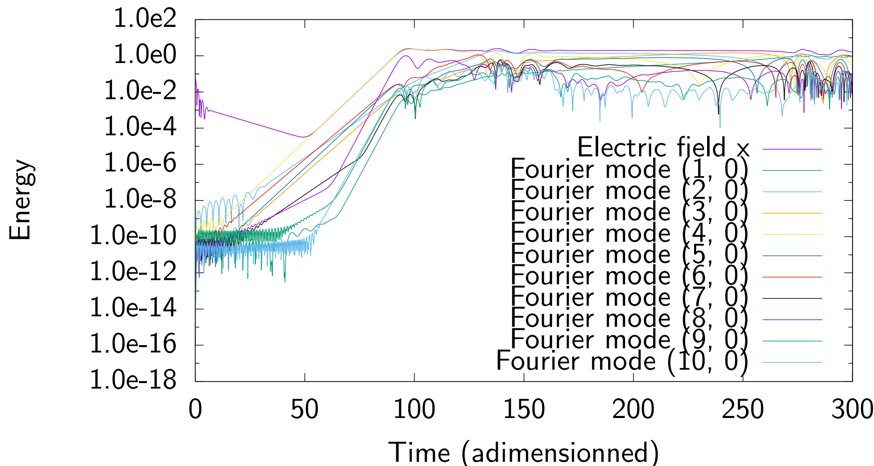
Weak scaling on Curie, for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations).

$$\begin{cases} f^0(\vec{v}) = \dfrac{v_x^2 e^{-\frac{|\vec{v}|^2}{2}}}{2\pi} \\ f(\vec{x}, \vec{v}, 0) = (1 + A\cos(\frac{y}{2}) + A\cos(\frac{x+y}{2})) \times f^0(\vec{v}), A = 0.001 \end{cases}$$



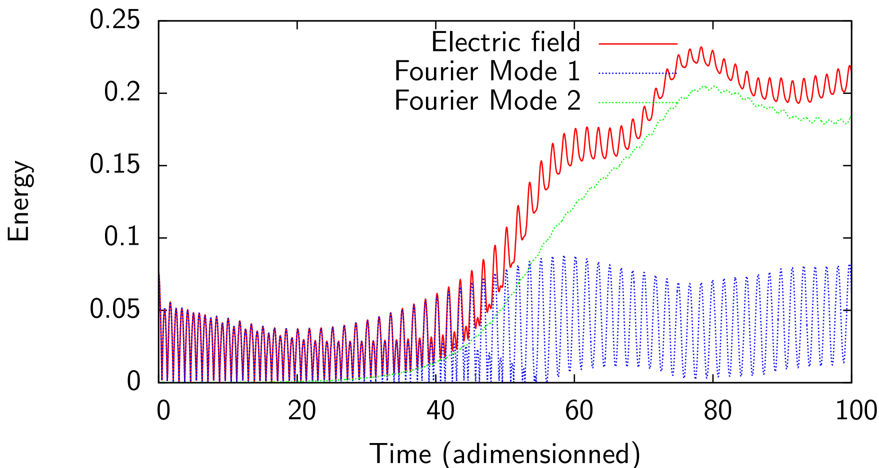
≪ Multi ≫-Modes : Beware of the Numerical Errors

$$\begin{cases} f^0(\vec{v}) = \dfrac{v_x^2 e^{-\frac{|\vec{v}|^2}{2}}}{2\pi} \\ f(\vec{x}, \vec{v}, 0) = (1 + A\cos(11 \times 0.1 \times x)) \times f^0(\vec{v}), A = 0.01 \end{cases}$$



Two Species : Bilinear Instability 1d1v

$$\begin{cases} f_e(x,v,0) = \dfrac{e^{-\frac{v^2}{2}}}{\sqrt{2\pi}} \quad \text{and} \quad f_i(x,v,0) = 8\dfrac{e^{-2v^2}}{\sqrt{2\pi}}(1 + A\cos(kx)) \\ \varepsilon = \sqrt{0.1}, A = 0.01, k = \dfrac{2\pi}{21} \approx 0.299 \end{cases}$$



Conclusions - Outlook

- efficient PIC code with standard numerical schemes
  - optimizations from other papers
  - −36% in cache misses (space-filling curves)
  - −31% in the update-positions loop (enhanced vectorization)
- unified framework for PIC and Semi-Lagrangian codes
- to be tested
  - use an AoS in memory, transpose a little portion of the data (before the computations) to benefit from unit stride vectorization (cf. Bowers *et al.* 2003)
  - 3d3v model for which we expect better speedups in the accumulate loop (cf. Vincenti *et al.* 2016)
  - auto-tuning of the sorting (on any architecture) for long simulations
  - port the code to Many Integrated Cores architectures

ybarsamian@unistra.fr