Pic-Vert: A Particle-in-Cell Implementation for Multi-Core Architectures

Yann Barsamian

Université de Strasbourg

Laboratoire des sciences de l'Ingénieur, de l'Informatique et de l'Imagerie (ICube), CNRS UMR 7357

INRIA Nancy

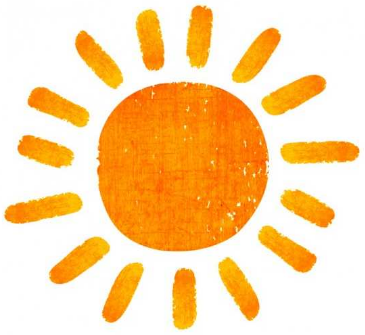
October 2018

Today's Outline

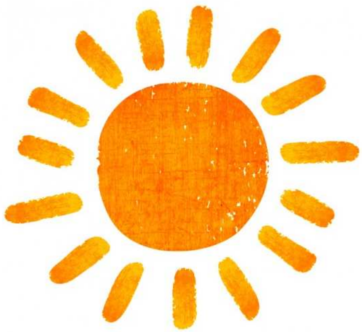
- Introduction
  - The context of this thesis.
  - Some computer architecture.
  - The Particle-in-Cell model.
- (Some) contributions
  - In the context of the standard algorithm.
  - With the help of a data structure crafted for our needs.
  - Comparison to the state-of-the-art.
- Conclusion
  - Other contributions.
  - Summary.
  - Perspectives.

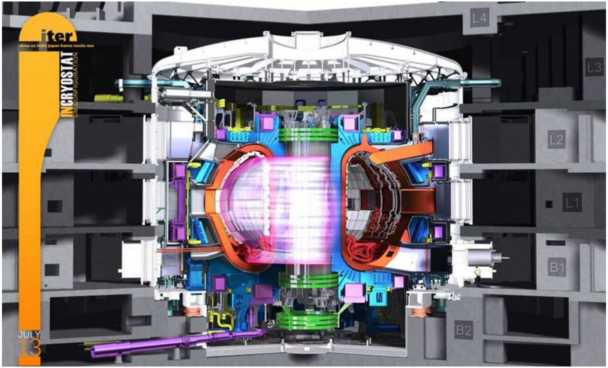
Introduction

Step 1.



Step 2.



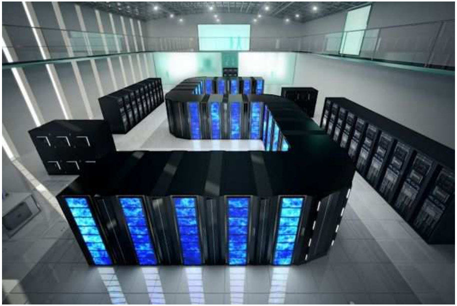


Step 3. ITER[1] tokamak[2]

(also applicable in other contexts, *e.g.*, astrophysics, where we have to model different particles / planets / ... that interact)

[1]"The way" (in Latin) to produce energy (Cadarache, France)

[2]Токамак: тороидальная камера с магнитными катушками (toroidal chamber with magnetic coils)



Marconi

Rank	Country	Cores
1		2 282 544
2		10 649 600
5		391 680
6		361 760
11		570 020
13		253 600
14		561 408

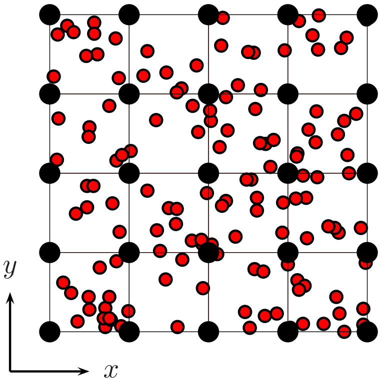
During this thesis, we used:

- Occigen (, 85 824 cores, Rank 70),
- Marconi (, 54 432 cores, Rank 98),
- Curie (, 77 184 cores, Rank 145), and
- Icps-gc-6 (2×10 cores) and this laptop (2 cores).

Source: June 2018 list of `https://www.top500.org`.

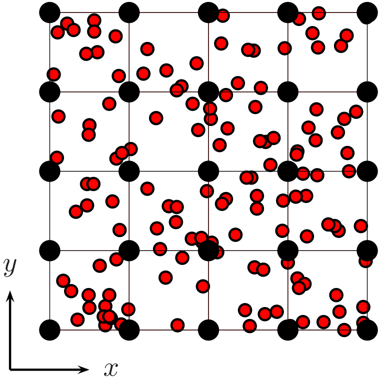
$$\begin{cases} \dfrac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f - \vec{E} \cdot \nabla_{\vec{v}} f = 0 & \text{Vlasov} \\ \nabla_{\vec{x}} \vec{E} = \rho = 1 - \displaystyle\int f \, d\vec{v} & \text{Poisson} \end{cases}$$

- Distribution function f: N numerical particles (red)
- Electric field $\vec{E}$ and charge density ρ: 3d grids (black)



$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v} \quad \text{and} \quad \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = -\vec{E} & \text{Vlasov characteristics} \\ \nabla_{\vec{x}} \vec{E} = \rho = 1 - \int f \; d\vec{v} & \text{Poisson} \end{cases}$$

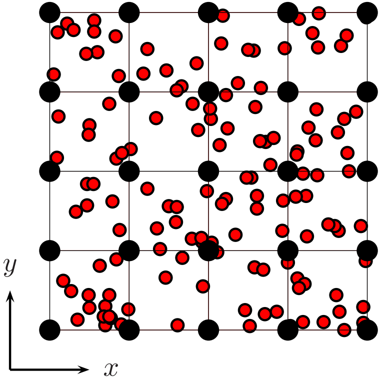
- Distribution function f: N numerical particles (red)
- Electric field $\vec{E}$ and charge density ρ: 3d grids (black)



Kinetic Modeling with Particle-in-Cell (PIC) Methods

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v} \quad \text{and} \quad \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = -\vec{E} & \text{Vlasov characteristics} \\ \nabla_{\vec{x}} \vec{E} = \rho = 1 - \displaystyle\int f \, d\vec{v} & \text{Poisson} \end{cases}$$

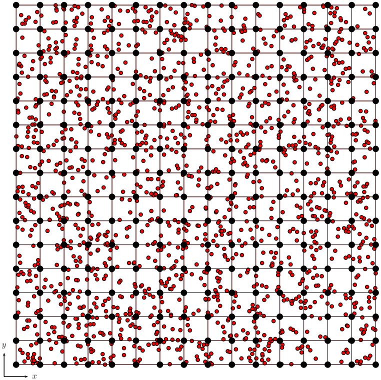
- Distribution function f: N numerical particles (red)
- Electric field $\vec{E}$ and charge density ρ: 3d grids (black)



- Physical effects on small scale (+ large scale)
- Noise (numerical errors when N is small)
- Frequent particle motion

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v} \quad \text{and} \quad \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = -\vec{E} & \text{Vlasov characteristics} \\ \nabla_{\vec{x}} \vec{E} = \rho = 1 - \displaystyle\int f \, d\vec{v} & \text{Poisson} \end{cases}$$

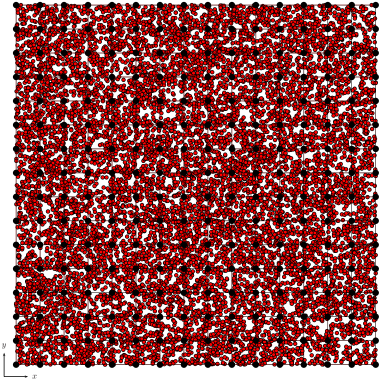
- Distribution function f: N numerical particles (red)
- Electric field $\vec{E}$ and charge density ρ: 3d grids (black)



- Physical effects on small scale (+ large scale)
  $\Rightarrow$ increase $ncx \times ncy \times ncz$
  ($1\ 000 \times 1\ 000 \times 1\ 000$)
- Noise (numerical errors when N is small)
- Frequent particle motion

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v} \quad \text{and} \quad \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = -\vec{E} & \text{Vlasov characteristics} \\ \nabla_{\vec{x}} \vec{E} = \rho = 1 - \displaystyle\int f \, d\vec{v} & \text{Poisson} \end{cases}$$

- Distribution function f: N numerical particles (red)
- Electric field $\vec{E}$ and charge density ρ: 3d grids (black)



- Physical effects on small scale (+ large scale)
  $\Rightarrow$ increase $ncx \times ncy \times ncz$
  $(1\ 000 \times 1\ 000 \times 1\ 000)$
- Noise (numerical errors when N is small)
  $\Rightarrow$ increase $\frac{N}{ncx \times ncy \times ncz}$
  (10 000 to 1 000 000)
- Frequent particle motion

Particle-in-Cell (PIC) Pseudo-Code

Initialization:

Initialize N particles   `icell, d{x,y,z}, v{x,y,z}` of size `[N]`
Compute ρ and E `rho, E{x,y,z}` of size `[ncx][ncy][ncz]`

Algorithm:

Foreach time iteration **do**
    If (*condition*) **then**
        Sort the particles[3] $\mathcal{O}(N)$ counting sort
    End If
    Set all cells of ρ to 0
    Foreach particle **do**
        Update the velocity $v+=-E\Delta t$
        Update the position $x+=v\Delta t$
        Deposit the charge on the nearest ρ cells
    End Foreach
    Compute E from ρ FFT Poisson solver
End Foreach

[3] Decyk, Karmesin, de Boer, & Liewer (1996)

Particle-in-Cell (PIC) Pseudo-Code

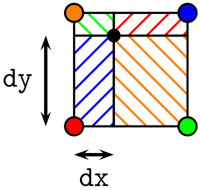
```
Initialization:
Initialize N particles    icell, d{x,y,z}, v{x,y,z} of size [N]
Compute ρ and E           rho, E{x,y,z} of size [ncx][ncy][ncz]

Algorithm:                                          Execution time breakdown
Foreach time iteration do
    If (condition) then
        Sort the particles[3]                                          10%[4]
    End If
    Set all cells of ρ to 0
    Foreach particle do
        Update the velocity                                            50%[4]
        Update the position                                            25%[4]
        Deposit the charge on the nearest ρ cells                      15%[4]
    End Foreach
    Compute E from ρ                                                   <1%[4]
End Foreach
```

[3] Decyk, Karmesin, de Boer, & Liewer (1996)

[4] Any difference in system hardware or software design or configuration may affect actual performance (-:

The Cloud-in-Cell[5] method:

$$v+=-E\Delta t$$




```
for (i = 0; i < N; i++) {
    vx[i] -= delta_t * (
               (    dx[i])*(    dy[i])*E[i_cell[i]].x_ne
             + (1.-dx[i])*(    dy[i])*E[i_cell[i]].x_nw
             + (    dx[i])*(1.-dy[i])*E[i_cell[i]].x_se
             + (1.-dx[i])*(1.-dy[i])*E[i_cell[i]].x_sw);
    vy[i] -= delta_t * (
               (    dx[i])*(    dy[i])*E[i_cell[i]].y_ne
             + (1.-dx[i])*(    dy[i])*E[i_cell[i]].y_nw
             + (    dx[i])*(1.-dy[i])*E[i_cell[i]].y_se
             + (1.-dx[i])*(1.-dy[i])*E[i_cell[i]].y_sw);
}
```

[5] Birdsall & Fuss (1969)

Contributions (part I)

[i] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell Code". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, 2017, pp. 1168–1177. DOI: 10.1109/IPDPSW.2017.74

[ii] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Layouts for a Three-Dimensional Electrostatic Particle-in-Cell Code". In: *Journal of Computational Science* 27 (2018), pp. 345–356. DOI: 10.1016/j.jocs.2018.06.004.

	Sort	Upd. v	Upd. x	Deposit	Total
Do not sort	0.0	98.0	64.6	35.9	199.0
Sort every 100	3.6	78.3	64.4	25.6	177.0
Always sort	209.0	66.3	64.2	13.4	353.0

Execution time (in s). Test case: 200 000 000 particles, 128×128 grid, $\Delta t = 0.1$, 500 iterations. Architecture: Intel Broadwell, 18 cores, 76.8 GB/s.

Periodic sorting: better data locality, and shorter overall time: find the best frequency[6].

Sorting at each iteration[7]: enhancement of the data locality & vectorization of the update velocities loop, but too costly.

Magic sorting that lasts longer[8]: needs less frequent sorting.

[6] Marin, Jin, & Mellor-Crummey (2008)

[7] Lanti, Tran, Jocksch, Hariri, Brunner, Gheller, & Villard (2016)

[8] Barsamian, Hirstoaga, & Violard (2017); Barsamian, Hirstoaga, & Violard (2018)

	T (s)	%	Acc. %
Baseline[9]	120.4	0.0	0.0
+ Loop not-so-invariant-code-motion	113.4	5.8	5.8
+ Loop Fission	97.9	13.7	18.7
+ Redundant arrays (E, ρ)[10,11]	94.0	4.0	21.9
+ Structure of Arrays (*particles*)	76.0	19.1	36.9
+ Space-filling curves (E, ρ)	72.6	4.5	39.7
+ Optimized update-positions	68.8	5.2	42.8

Total execution time, gains and accumulated gains, for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 20 iterations). Architecture: Intel Haswell.

[9] Chacon-Golcher, Hirstoaga, & Lutz (2016), `http://selalib.gforge.inria.fr`
[10] Bowers (2003)
[11] Vincenti, Lobet, Lehe, Sasanka, & Vay (2016)

Overall Optimization Gains

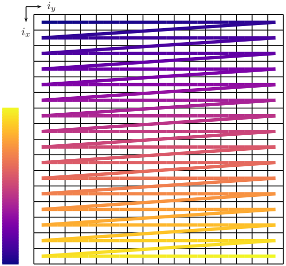
	T (s)	%	Acc. %
Baseline[9]	120.4	0.0	0.0
+ Loop not-so-invariant-code-motion	113.4	5.8	5.8
+ Loop Fission	97.9	13.7	18.7
+ Redundant arrays (E, ρ)[10,11]	94.0	4.0	21.9
+ Structure of Arrays (*particles*)	76.0	19.1	36.9
+ Space-filling curves (E, ρ)	72.6	4.5	39.7
+ Optimized update-positions	68.8	5.2	42.8

Total execution time, gains and accumulated gains, for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 20 iterations). Architecture: Intel Haswell.

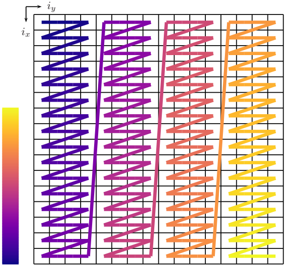
75 million particles processed/second on one core.

[9] Chacon-Golcher, Hirstoaga, & Lutz (2016), `http://selalib.gforge.inria.fr`
[10] Bowers (2003)
[11] Vincenti, Lobet, Lehe, Sasanka, & Vay (2016)

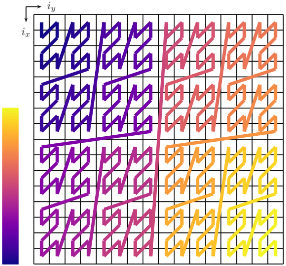
Space-Filling Curves for E and ρ: in 2d



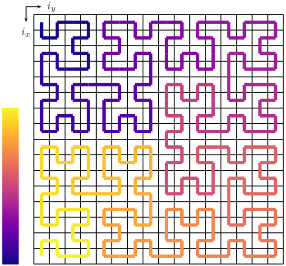
Row-Major (Standard layout in C)



L4D curve (Chatterjee, Jain, Lebeck, Mundhra, & Thottethodi, 1999)

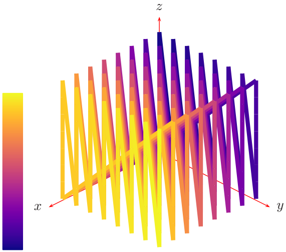


Morton curve (Morton, 1966)

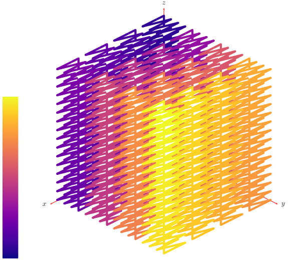


Hilbert curve (Hilbert, 1891)

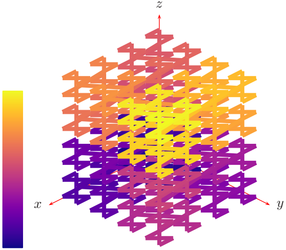
Space-Filling Curves for E and ρ: in 3d



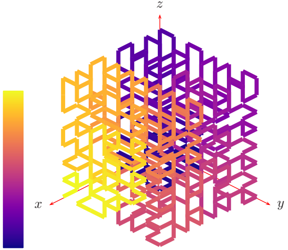
Row-Major (Standard layout in C)



L6D curve
(Barsamian, Hirstoaga, & Violard, 2018)



Morton curve (Morton, 1966)



Hilbert curve (Hilbert, 1891)

> *These layout functions are inappropriate for programs that access array elements randomly. [...] Even more than for the 4D layout, the Morton layout function is expensive to compute naively.*
>
> Chatterjee *et al.*, 1999

> *This paper investigates using data and computation reorderings to improve [...] irregular applications.*
>
> Mellor-Crummey, Whalley, & Kennedy, 2001

> *As Morton-order representation [...] attracts more users because of its excellent block locality, the efficiency of these conversions becomes important.*
>
> Raman & Wise, 2007

Space-Filling Curves for E and ρ: Results

	Update v	Update x	Deposit	Sort	Total
Row-major	63.6	39.7	42.8	28.6	177
L4D	57.5	40.0	32.0	28.6	161
Morton	59.3	39.8	29.8	28.4	160
Hilbert	59.0	323.7	33.6	28.6	452

	Update v	Update x	Deposit	Sort	Total
Row-major	92.6	55.3	31.5	21.4	202
L6D	85.5	55.5	29.9	20.9	193
Morton	89.4	56.7	33.5	19.8	200
Hilbert	87.3	244.4	29.2	20.3	382

Time spent in the different loops (in seconds) when using several space-filling curves.
Top: 2d, 512 x 512 grid, 1 billion particles, 100 iterations (sorting every 20 iterations).
Bottom: 3d, 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations).

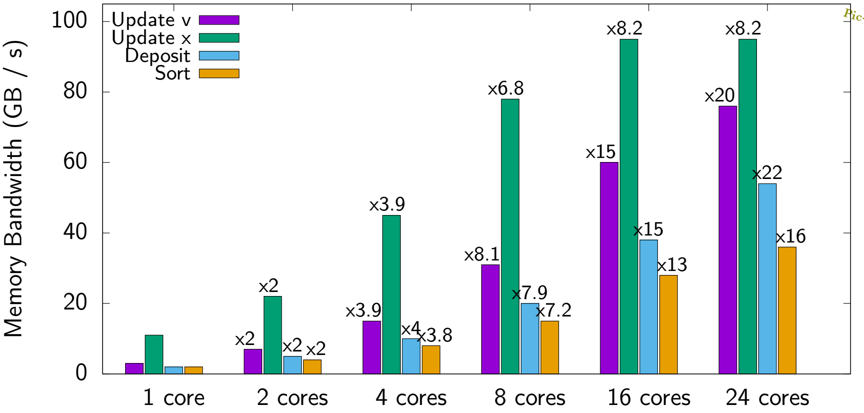
Loop not-so-invariant-code-motion

```
// Update-velocities
for (size_t i = 0; i < num_particle; i++) {
    vx[i] -= delta_t * E_x_part; // Reads array Ex
    vy[i] -= delta_t * E_y_part; // Reads array Ey
}
```

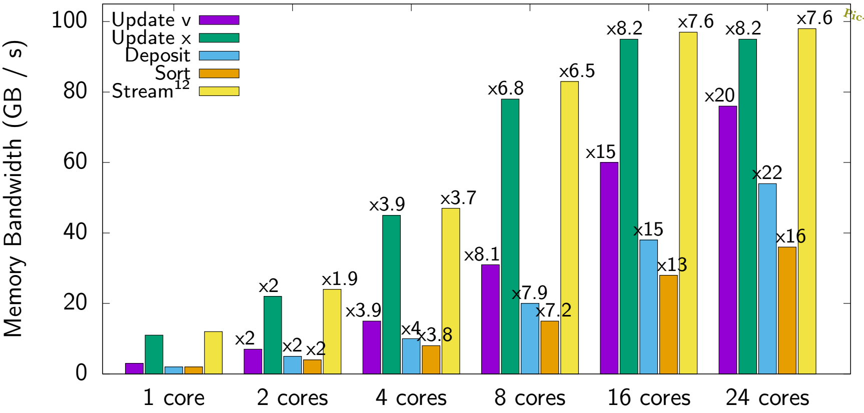
Loop not-so-invariant-code-motion

```
void poisson_solver([...] double** Ex, double** Ey) {
    [...]
    for (size_t i = 0; i < ncx; i++) {
        for (size_t j = 0; j < ncy; j++) {
            Ex[i][j] *= delta_t;
            Ey[i][j] *= delta_t;
        }
    }
}

// Update-velocities
for (size_t i = 0; i < num_particle; i++) {
    vx[i] -= E_x_part; // Reads array Ex
    vy[i] -= E_y_part; // Reads array Ey
}
```

Strong Scaling on 24 Cores, With Loop Fission



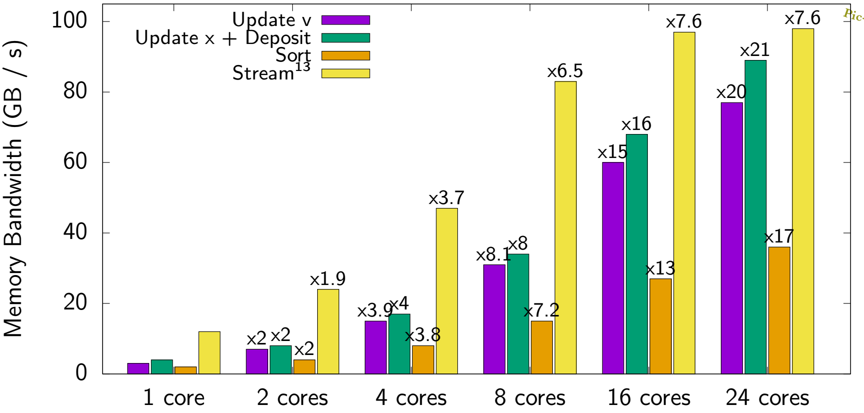
Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.



Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.

[12] McCalpin (1995) - Code v5.10 (2013)

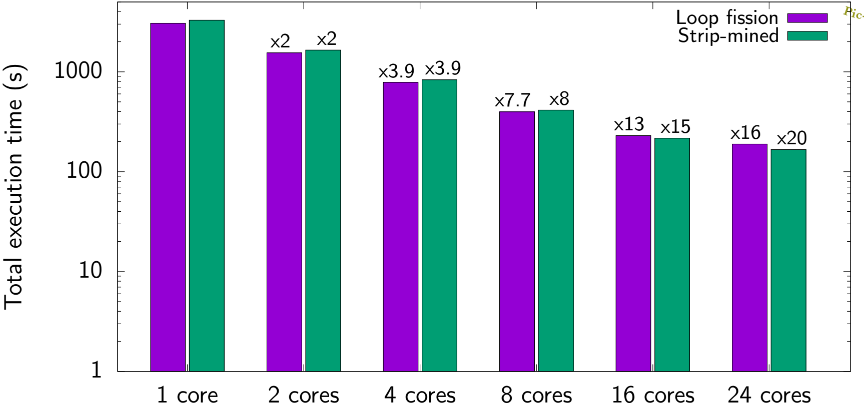
Strong Scaling on 24 Cores, With Strip-Mining



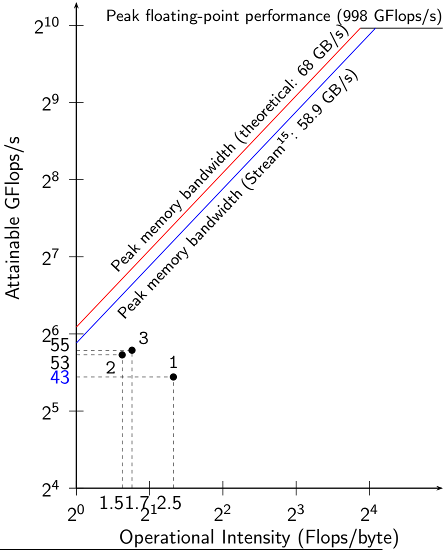
Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.

[13] McCalpin (1995) - Code v5.10 (2013)

Strong Scaling on 24 Cores, With Strip-Mining



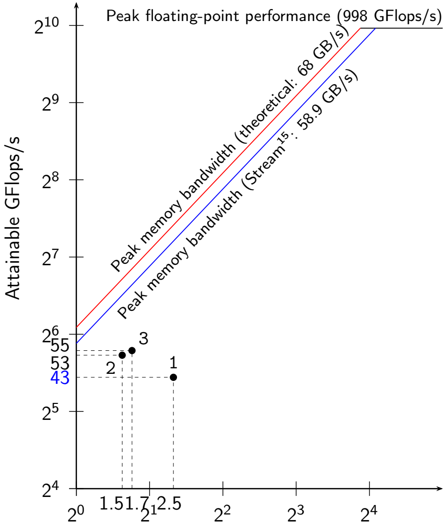
Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.
Strip-mined is 6.7% slower on 1 core but 12% faster on 24 cores.



- Baseline (1): missing computational efficiency.
- Loop fission (2): missing memory efficiency.
- Loop strip-mining (3): the best of two worlds.

[14] Williams, Waterman, & Patterson (2009)
[15] McCalpin (1995) - Code v5.10 (2013)

Roofline Model[14] on Intel Haswell (12 Cores)



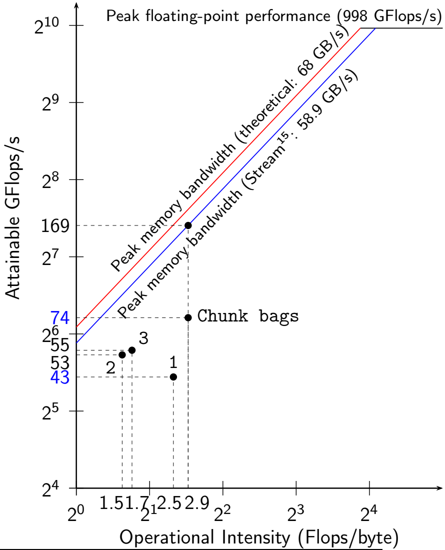
- Baseline (1): missing computational efficiency.
- Loop fission (2): missing memory efficiency.
- Loop strip-mining (3): the best of two worlds.

Can we do better?

[14] Williams, Waterman, & Patterson (2009)

[15] McCalpin (1995) - Code v5.10 (2013)

Roofline Model[14] on Intel Haswell (12 Cores)



- Baseline (1): missing computational efficiency.
- Loop fission (2): missing memory efficiency.
- Loop strip-mining (3): the best of two worlds.

Can we do better?

[14] Williams, Waterman, & Patterson (2009)
[15] McCalpin (1995) - Code v5.10 (2013)

Contributions (part II)

[iii] Y. Barsamian, A. Charguéraud, and A. Ketterlin. "A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell Method". In: *Parallel Processing and Applied Mathematics: 12th International Conference (PPAM)*. vol. 10777. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 133–144.
DOI: 10.1007/978-3-319-78024-5_13

[iv] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. "Efficient Strict- Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors". In: *24th International Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 11014. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 749–763.
DOI: 10.1007/978-3-319-96983-1_53

[v] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. *Software artifacts for Euro-Par 2018 paper: "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors"*. Figshare. 2018.
URL: https://doi.org/10.6084/m9.figshare.6391796.

To sort or not to sort?

	Sort	Upd. v	Upd. x	Deposit	Total
Do not sort	0.0	98.0	64.6	35.9	199.0
Sort every 100	3.6	78.3	64.4	25.6	177.0
Always sort	209.0	66.3	64.2	13.4	353.0

Execution time (in s). Test case: 200 000 000 particles, 128 $\times$ 128 grid, $\Delta t = 0.1$, 500 iterations. Architecture: Intel Broadwell, 18 cores, 76.8 GB/s.

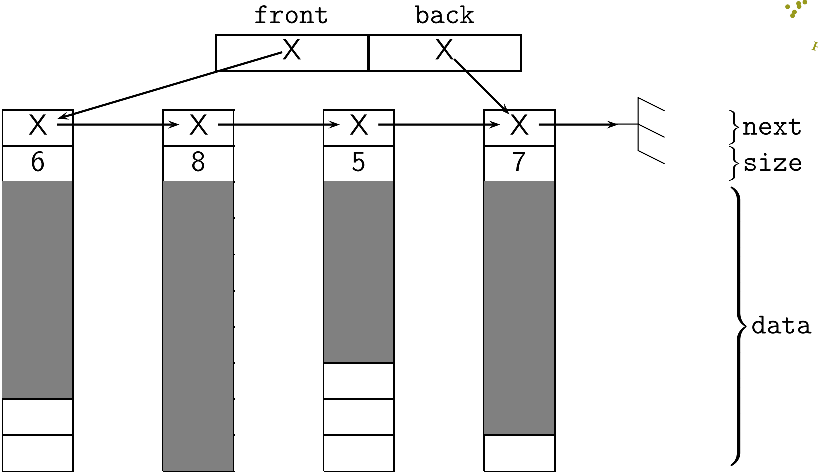
Periodic sorting: better data locality, and shorter overall time: find the best frequency.

Sorting at each iteration: enhancement of the data locality & vectorization of the update velocities loop, but too costly.

~~Magic sorting that lasts longer: needs less frequent sorting.~~ Efficient data structure to keep particles sorted[16]: avoid the sorting step.

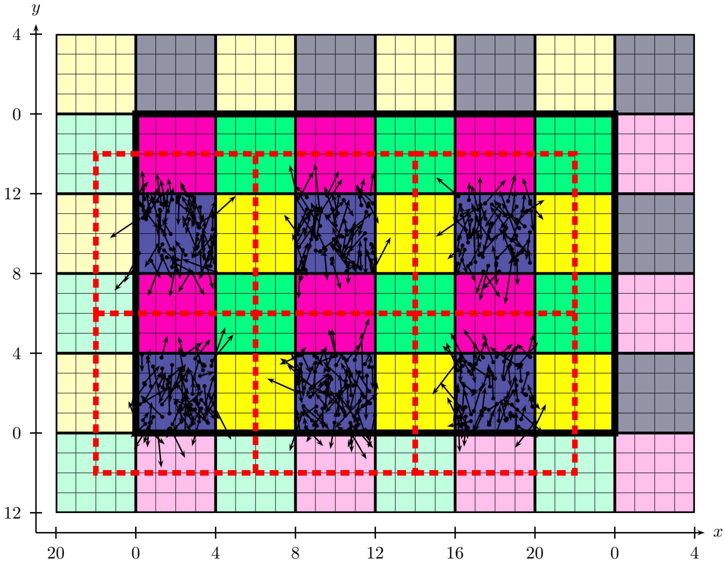
[16]Durand, Raffin, & Faure (2012); Nakashima, Summura, Kikura, & Miyake (2017); Barsamian, Charguéraud, & Ketterlin (2017)

Chunk Bags: Linked Lists of Fixed-Size Arrays



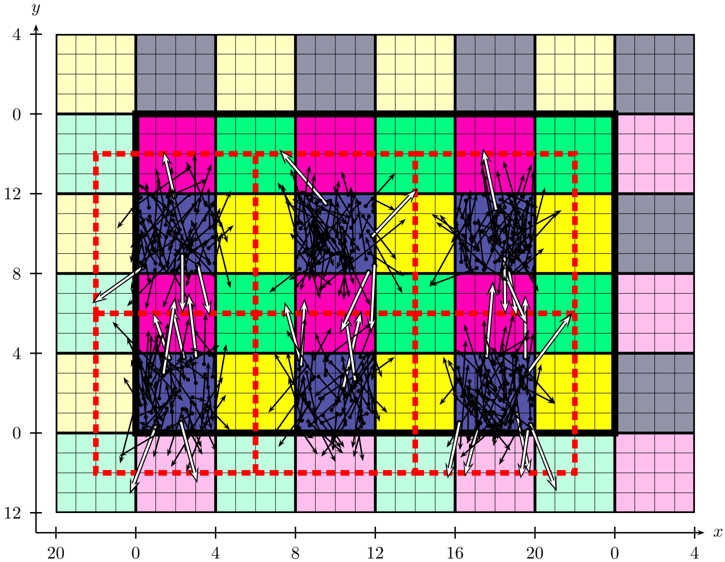

```
struct chunk { struct chunk* next; int size; // 0<=size<=K
               float  dx[K], dy[K], dz[K];
               double vx[K], vy[K], vz[K]; } chunk;
struct { chunk* front, back; } bag;
```

The Eight-Colors Algorithm[17]



8 phases to tame the number of data races when moving particles.

[17] Kong, Huang, Ren, & Decyk (2011)

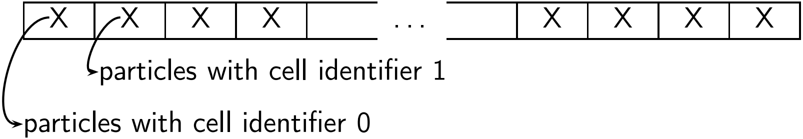


Particles moving more than half a tile away require special care.

[17] Kong, Huang, Ren, & Decyk (2011)

Chunk Bags: Particle Arrays

```
chunkbag particles[nbCells] // nbCells = ncx*ncy*ncz
```



```
chunkbag particlesNextPrivate[nbCells],
         particlesNextShared[nbCells]
```

- `particlesNextPrivate[i]` receives particles moving to a nearby cell i: no atomic operation required.
- `particlesNextShared[i]` receives particles moving to a remote cell i: atomic push used.
- `particles[i]` at the next time step is obtained by merging the two.

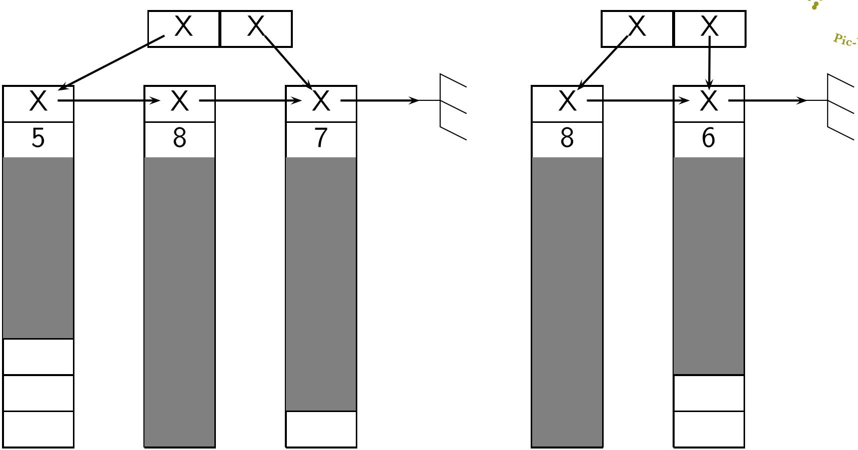
Chunk Bags: Code

```
void bag_push_serial([...]) {
  chunk* c;
  int id;

    c = b->front; // The front chunk.

    id = c->size++; // The id of the last free cell.

      c->dx[id] = dx; c->dy[id] = dy; c->dz[id] = dz;
      c->vx[id] = vx; c->vy[id] = vy; c->vz[id] = vz;
      if (id == CHUNK_SIZE - 1) // The chunk is now full.
        add_front_chunk(b, thread_id);

}
```

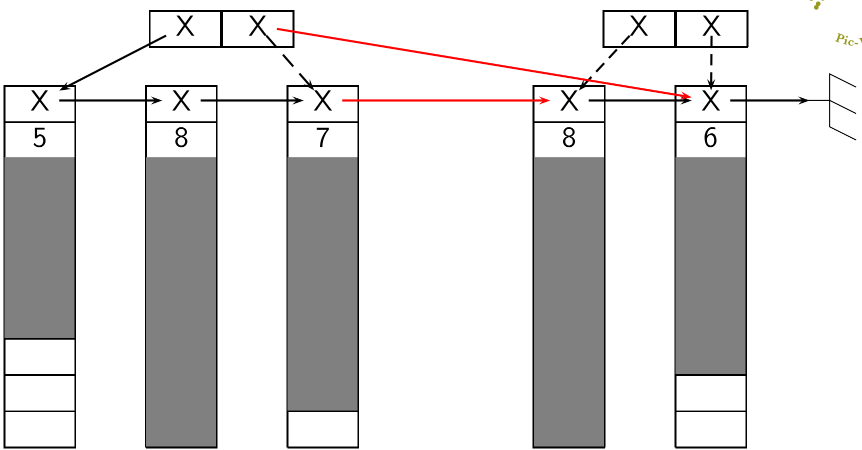
Chunk Bags: Code

```
void bag_push_concurrent([...]) {
  chunk* c;
  int id;
  while (true) { // Until success.
    c = b->front; // The front chunk.
    #pragma omp atomic capture
    id = c->size++; // The id of the last free cell.
    if (id < CHUNK_SIZE) { // The chunk was not full.
      c->dx[id] = dx; c->dy[id] = dy; c->dz[id] = dz;
      c->vx[id] = vx; c->vy[id] = vy; c->vz[id] = vz;
      if (id == CHUNK_SIZE - 1) // The chunk is now full.
        add_front_chunk(b, thread_id);
      return;
    } else { // The chunk was full.
      #pragma omp atomic write
      c->size = CHUNK_SIZE;
      while (atomic_read(&b->front) == c) {}
    }
  }
}
```

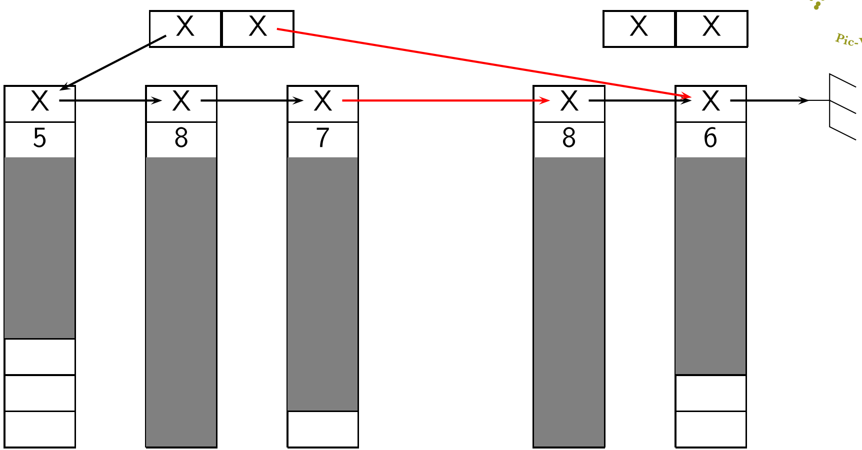
Chunk Bags: Merge Operation



Chunk Bags: Merge Operation



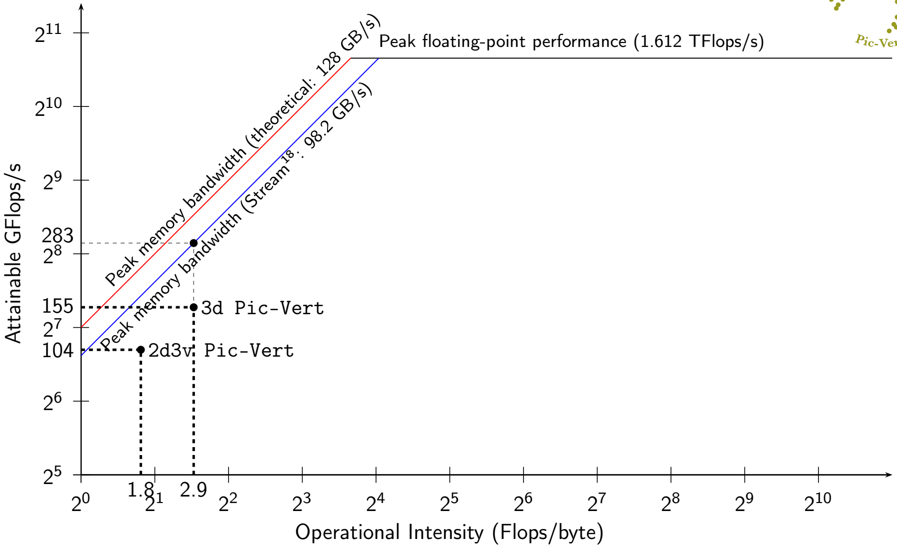
Chunk Bags: Merge Operation



Upper bound on the number of chunks: $\lceil N/K \rceil + 4 \cdot \text{nbCells}$.

All chunks allocated at initialization (no dynamic `malloc`/`free`).

Roofline Model[19] on Intel Skylake (24 Cores)



[18] McCalpin (1995) - Code v5.10 (2013)

[19] Williams, Waterman, & Patterson (2009)

Different implementations on different architectures: cores, memory bandwidth in GB/s, number of particles processed by second (absolute and normalized w.r.t. memory bandwidth).
Top: CPUs. Bottom: accelerators (GPUs, MICs).

Implem.	Architecture	Cores	M.B.	Part./s	Norm.
VPIC	IBM PowerXCell 8i	9	204.8	$173 \cdot 10^6$	0.85
OSIRIS	Intel Xeon E5-2680	8	51.2	$134 \cdot 10^6$	2.62
ORB5	Intel Xeon E5-2670	8	51.2	$69 \cdot 10^6$	1.35
PICADOR	Intel Xeon E5-2697 v3	14	68	$127 \cdot 10^6$	1.87
GTC-P	Intel Xeon E5 2692 v2	12	59.7	$100 \cdot 10^6$	1.68
PIConGPU	Intel Xeon E5-2698 v3	16	68	$111 \cdot 10^6$	1.63
Pic-Vert	Intel Xeon Platinum 8160	24	128	$740 \cdot 10^6$	5.78
Pic-Vert	Intel Xeon E5-2690 v3	12	68	$374 \cdot 10^6$	5.49
PIConGPU	NVIDIA Tesla GK210	2496	480	$336 \cdot 10^6$	0.70
ORB5	NVIDIA Tesla K20X	2688	250.0	$177 \cdot 10^6$	0.71
PICADOR	Intel Xeon Phi 7250 (KNL)	68	115.2	$298 \cdot 10^6$	2.59
EMSES	Intel Xeon Phi 7250 (KNL)	68	115.2	$1300 \cdot 10^6$	11.3

Fast-Moving Particles

A parameter that affects the efficiency of any PIC simulation is p, the fraction of particles that cross cell boundaries.

Most previous work focus on simulations with a low value of p.

- VPIC: the deposit step is only vectorized on 4 particles when none of those particles cross cell boundaries[20].
- UPIC: results are shown for p up to 12%.
- EMSES: the mechanism is shown to be efficient only when p is low (1–2% in this paper).

In our test cases, p reaches up to 99% (and particles move to arbitrarily far away cells).

By design, p has only little impact on our performance.

[20] With $p = 5\%$, this happens only $0.95^4 = 81\%$ of the time.

Experiments with different particle velocities, where the initial velocities follow the sum of two Gaussian distributions, like in the bump-on-tail instability:

$$f_0(\mathbf{x}, vx, vy, vz) = g(vx) \cdot g(vy) \cdot g(vz), \quad \text{with}$$
$$g(w) = \frac{1}{\sqrt{2\,\pi}\,v_{th}} \left(p_{\text{drift}} \exp\left(-\frac{(w - v_{\text{drift}})^2}{2v_{th}^2} \right) + (1 - p_{\text{drift}}) \exp\left(-\frac{w^2}{2v_{th}^2} \right) \right).$$

- up to 4.4% of "fast-moving particles" (more than 2 cells away),
- up to 3.7% of possible conflicts[21],
- if processed sequentially (EMSES): 85% slowdown on 24 cores[22],
- when processed with our shared bags: only 7.0% slowdown.

[21] Not all fast-moving particles go out of the "extended tile" — consider a particle on the far left of the tile moving 3 cells to the right.

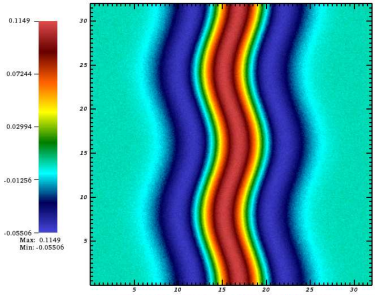
[22] Let t denote the single-core execution time. Assume 3.7% of sequential execution, and 96.3% using 24 cores. The parallel execution time is: $0.037t + 0.963t/24 = 1.85t/24$.

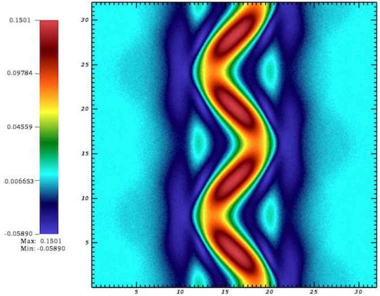
Conclusion

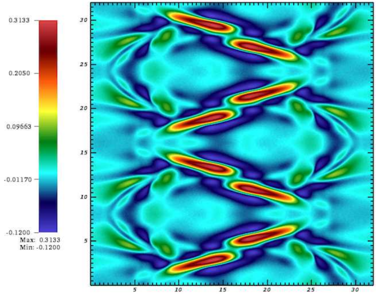
Validation: 2d3v Electron Hole Test Case[23]

- 64 billion particles,
- grid of size 512×512,
- time step 0.1,
- spatial domain $[0, 32]^2$,
- magnetic field 0.2.

Snapshots of ρ at t=0 (top right), 20 (bottom left), and 40 (bottom right).







[23] Muschietti, Roth, Carlson, & Ergun (2000)

Other Contributions

- Full advantage of vectorization (SIMD)
- Design of other variants for our algorithm with chunk bags
- Efficient 2d semi-Lagrangian implementation
- A common framework for Particle-in-Cell and semi-Lagrangian methods
- New 2d test cases with their theoretical analyses

[vi] Y. Barsamian, J. Bernier, S. A. Hirstoaga, and M. Mehrenberger. "Verification of $2D \times 2D$ and two-species Vlasov–Poisson solvers". In: *ESAIM: Proceedings and Surveys* 63 (2018), pp. 78–108.
DOI : 10.1051/proc/201863078.

[vii] Y. Barsamian, and M. Mehrenberger. "Semi-Lagrangian Simulations for Solving 2d2v Vlasov–Poisson Systems (one and two species)". In: *Platform for Advanced Scientific Computing (PASC), Minisymposium "Kinetic Simulations on HPC Platforms for Plasma Physics Applications (3/3): Parallelization and New Hardware"*. 2017.

Slides: slides_2017-06-27.pdf.

- Contributions
  - Particles sorted at all time with low memory overhead ($4 \cdot$ nbCells extra chunks, lowest in the state-of-the-art)
  - Optimal memory bandwidth usage: each particle is loaded from/written to main memory only once per iteration
  - Full advantage of SIMD
  - Efficient handling of fast particles
- Results on Intel Skylake, 24 cores, 128 GB/s
  - 740 million particles / second in 3d
  - 55% of the maximum bandwidth
- Comparison to state-of-the-art implementations thanks to a new metric

Future Outlooks

- Test our ideas when solving the Vlasov–Maxwell equations
- Test `Pic-Vert` on Many Integrated Core (MIC) architecture (more cores)
- Investigate the use of chunks in domain decomposition (distributed memory parallelism)
- Collaborate with other teams

http://www.barsamian.am/Pic-Vert/