A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell (PIC) Method

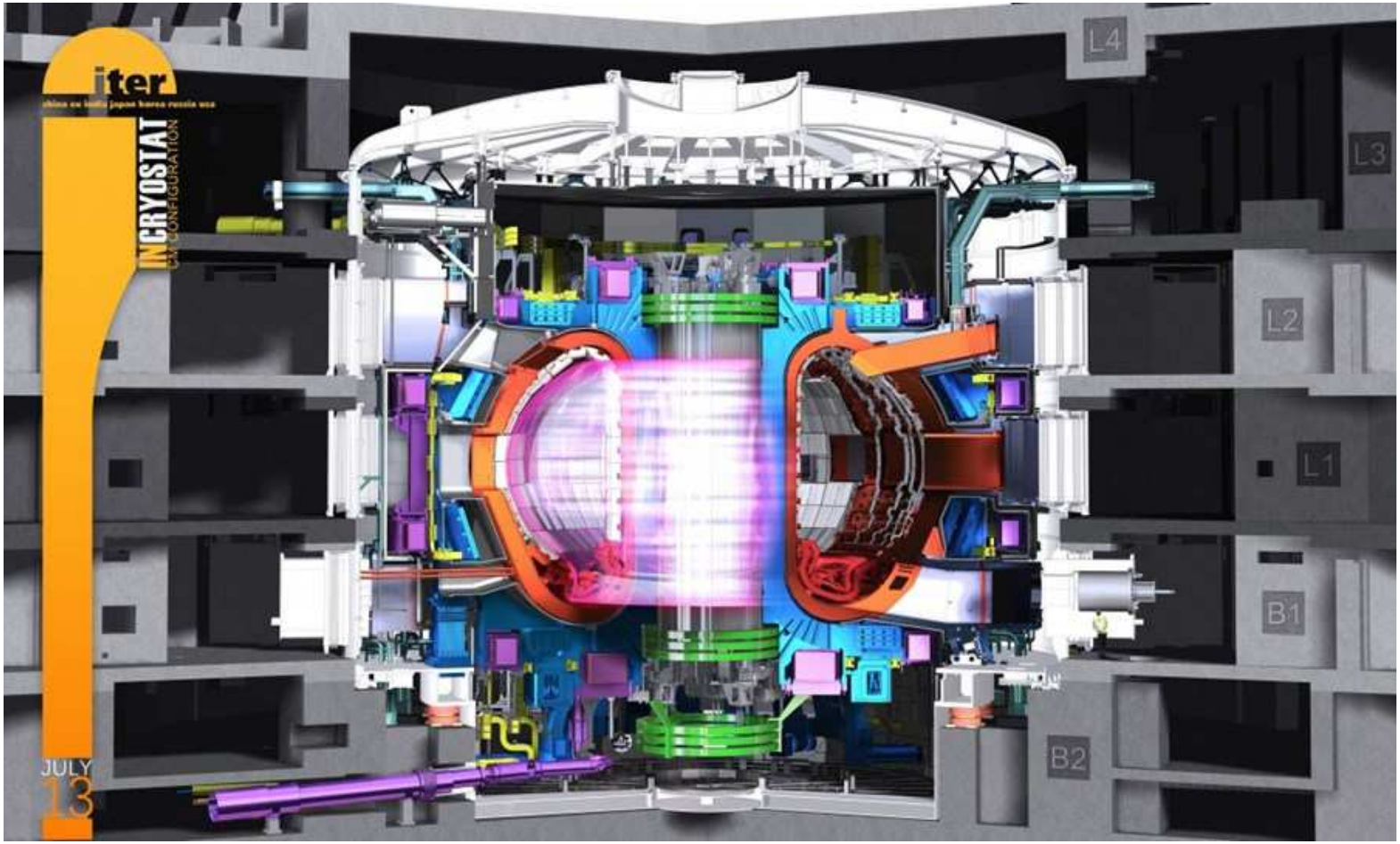
Yann Barsamian, Arthur Charguéraud, Alain Ketterlin

Université de Strasbourg

INRIA Nancy - Grand Est (CAMUS Team)
CNRS ICube Laboratory (ICPS Team)

PPAM 2017, Lublin (Poland)
September 2017

General Context



ITER[1] tokamak[2]: controlled thermonuclear fusion

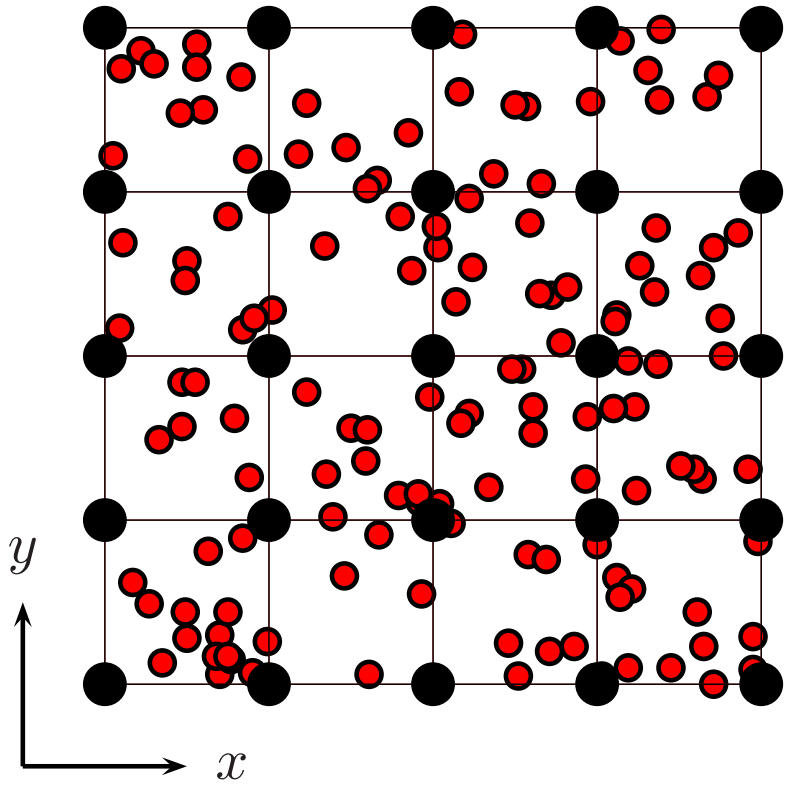
[1] "The way" (in Latin) to produce energy (Cadarache, France)

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

Kinetic Modeling with Particle-in-Cell Methods

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

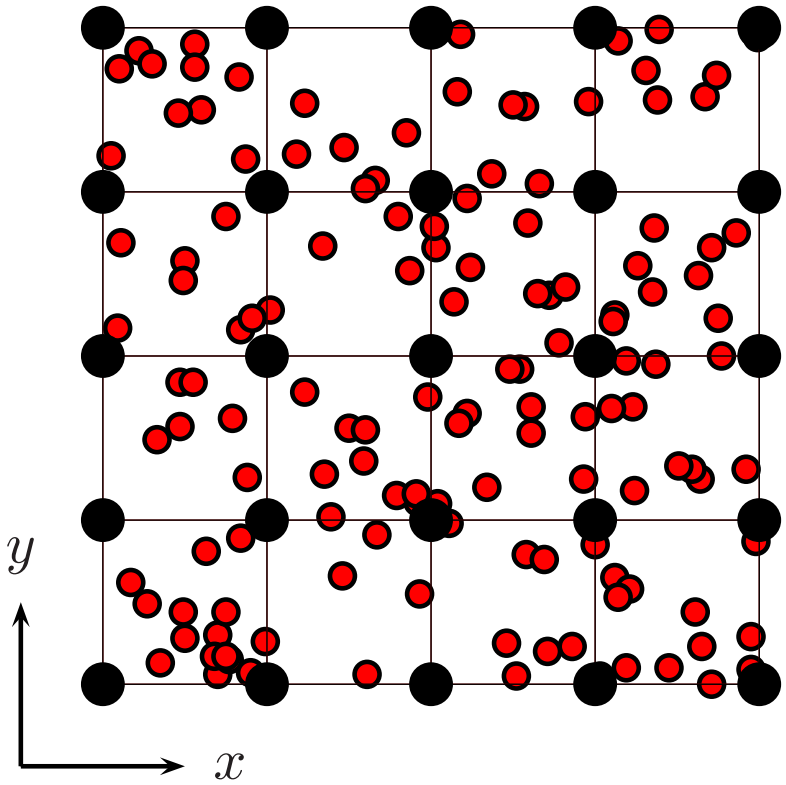
- distribution function f: N numerical particles (red)
- electric field $\vec{E}$ and charge density ρ: 2d grids (black)



Kinetic Modeling with Particle-in-Cell Methods

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

- distribution function f: N numerical particles (red)
- electric field $\vec{E}$ and charge density ρ: 2d grids (black)

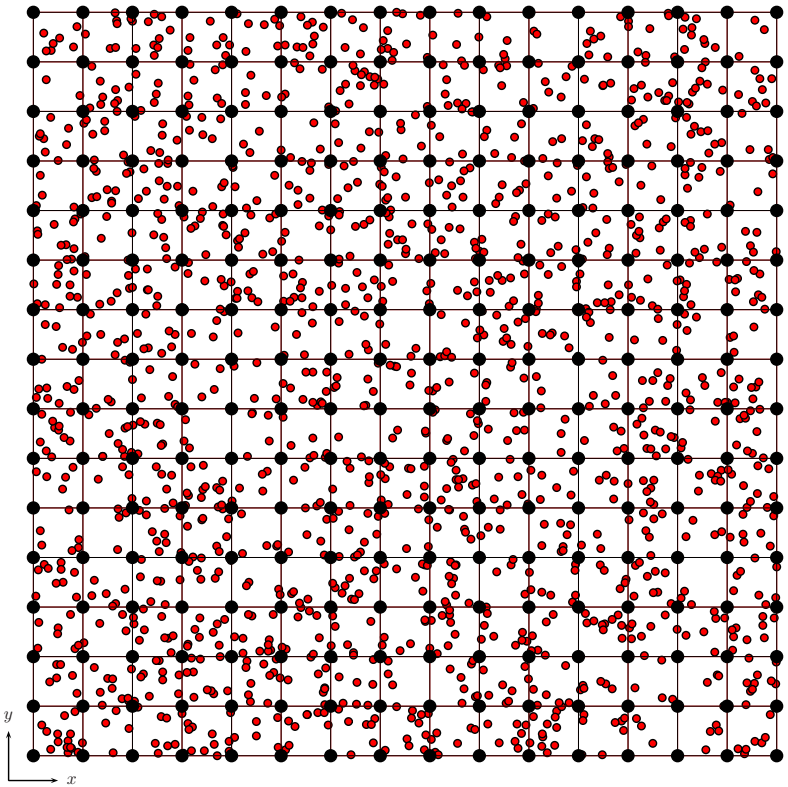


- Physical effects on small scale (+ large scale)
- Noise (numerical errors when N is small)
- Frequent particle motion

Kinetic Modeling with Particle-in-Cell Methods

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

- distribution function f: N numerical particles (red)
- electric field $\vec{E}$ and charge density ρ: 2d grids (black)

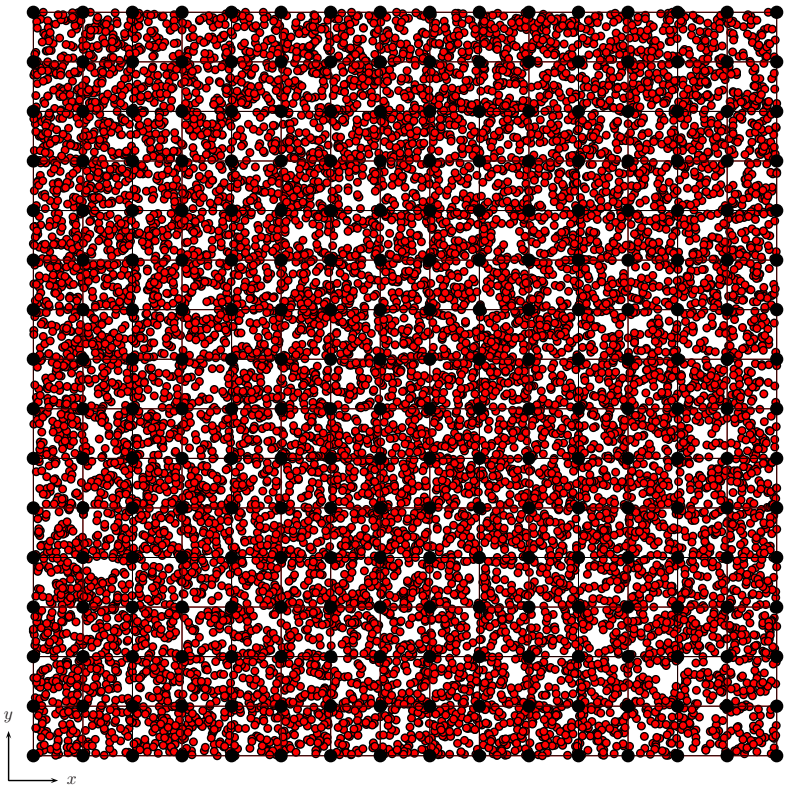


- Physical effects on small scale (+ large scale)
  $\Rightarrow$ increase $ncx \times ncy$ $(1,000 \times 1,000)$
- Noise (numerical errors when N is small)
- Frequent particle motion

Kinetic Modeling with Particle-in-Cell Methods

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

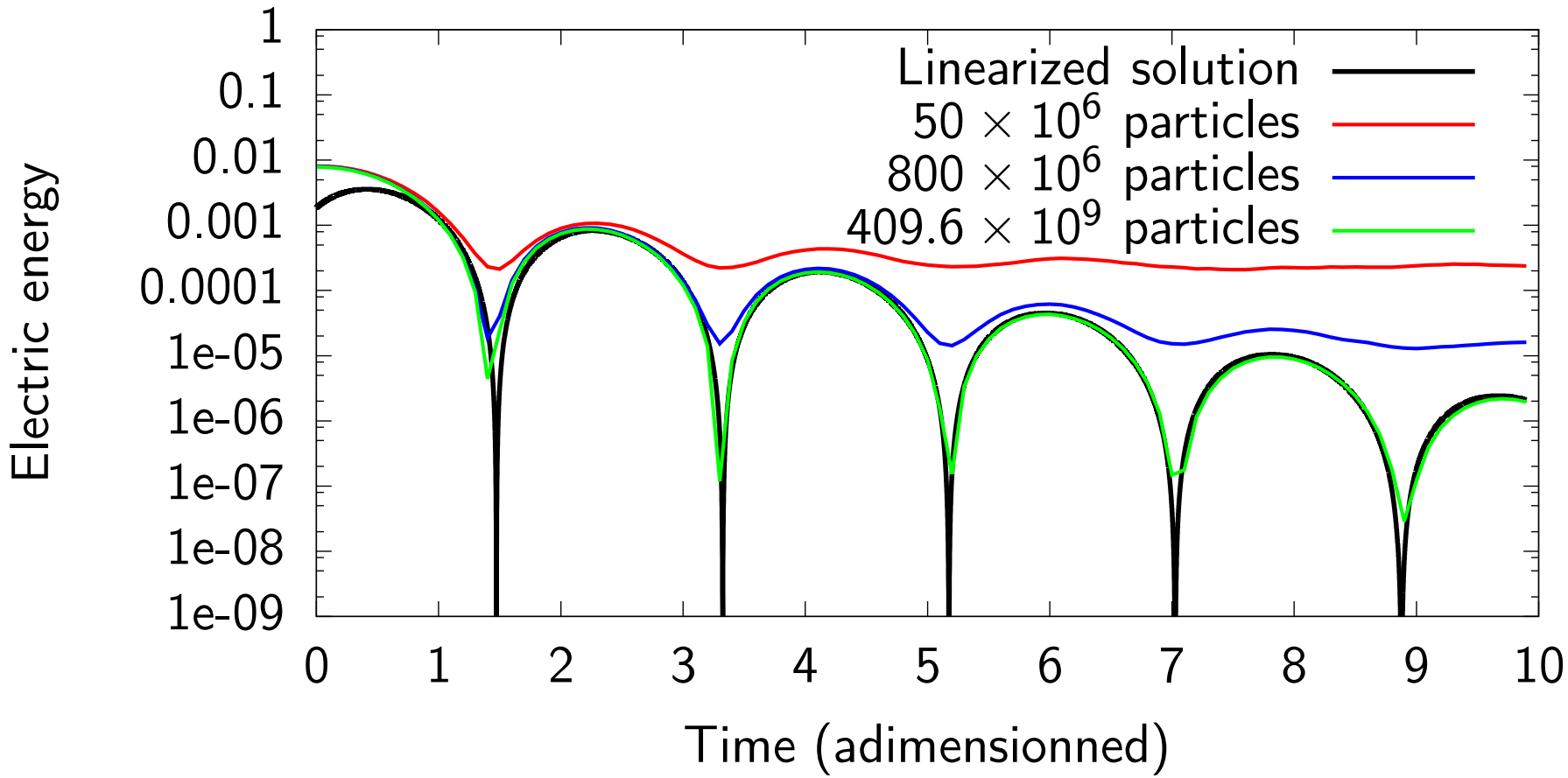
- distribution function f: N numerical particles (red)
- electric field $\vec{E}$ and charge density ρ: 2d grids (black)



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

Code Verification: Ландау (Landau) Damping

- basic test case with known mathematical approximate solution
- needs specific noise reduction techniques or a lot of particles



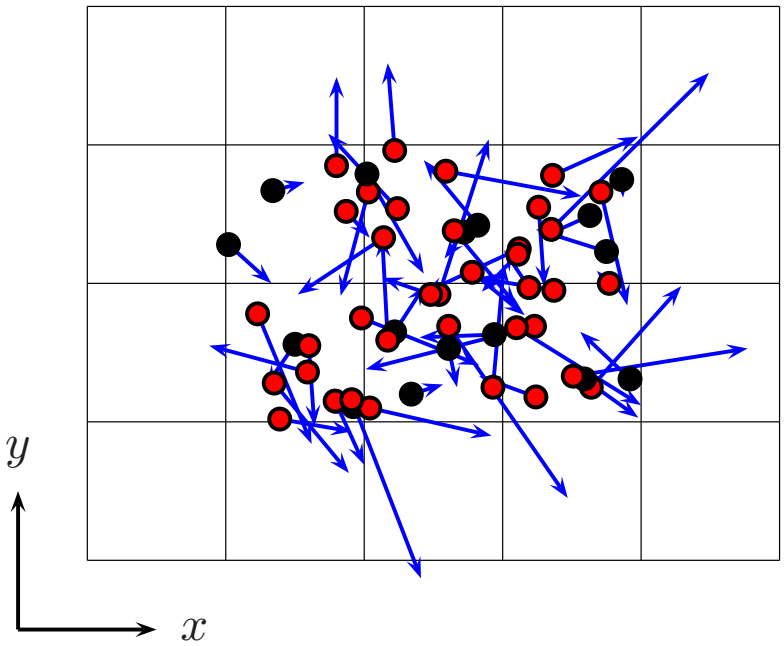
$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.1$, initial condition (Ландау, 1946):

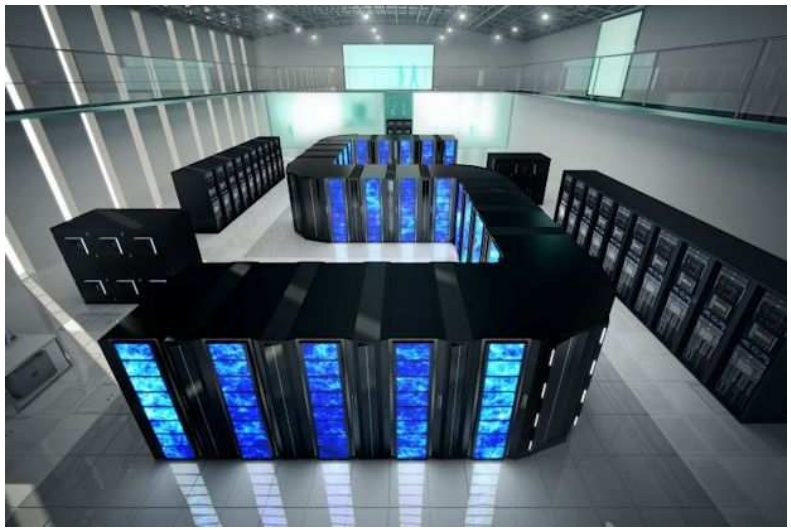
$$f(\overrightarrow{x}, \overrightarrow{v}, 0) = (1 + 0.01 \cos(x/2) \cos(y/2)) \, e^{-\frac{v_x^2 + v_y^2}{2}} / (2\pi)$$

So the only questions that remain are. . .

How to design an efficient code when particles move frequently? (up to 98% of particles change cell at each iteration)

How to derive a code that scales up well on thousands of cores, using MPI + OpenMP?





Marconi supercomputer

Particle-in-Cell Pseudo-Code

Initialization:

1 Initialize N particles — `icell, dx, dy, vx, vy` of size `[N]`

2 Compute ρ and E at $t = 0$ — `rho, Ex, Ey` of size `[ncx][ncy]`

Algorithm:

3 **Foreach** time iteration **do**

4 **If** (*condition*) **then**

5 Sort the particles[3] — $\mathcal{O}(N)$ counting sort

6 **End If**

7 Set all cells of ρ to 0

8 **Foreach** particle **do**

9 Update the velocity — $v+ = -E\Delta t$

10 Update the position — $x+ = v\Delta t$

11 Accumulate the charge on the nearest ρ cells

12 **End Foreach**

13 Compute E from ρ — FFT Poisson solver

14 **End Foreach**

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

Particle-in-Cell Pseudo-Code

Initialization:

1 Initialize N particles — `icell, dx, dy, vx, vy` of size `[N]`

2 Compute ρ and E at $t = 0$ — `rho, Ex, Ey` of size `[ncx][ncy]`

Algorithm: — Execution time breakdown

3 **Foreach** time iteration **do**

4 **If** (*condition*) **then**

5 Sort the particles[3] — 10%

6 **End If**

7 Set all cells of ρ to 0

8 **Foreach** particle **do**

9 Update the velocity — 40%

10 Update the position — 35%

11 Accumulate the charge on the nearest ρ cells — 15%

12 **End Foreach**

13 Compute E from ρ — <1%[4]

14 **End Foreach**

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

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

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

200,000,000 particles, 128×128 mesh, $\Delta t = 0.1$, 500 iterations.

Architecture: 18 threads, 4 memory channels, Intel Broadwell (2016).

Periodic sorting: better data locality, and shorter overall time. Best frequency?[5]

Sorting at each iteration[6]: enhancement of the data locality & vectorization of the update velocities loop, but too costly.

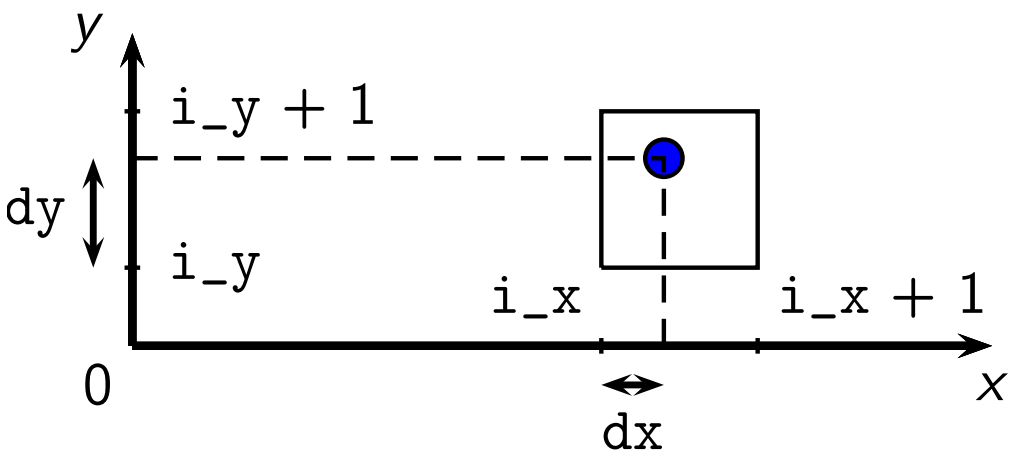
Efficient data structure to keep particles sorted[7,8]: avoid the sorting step.

[5] Dorobisz, Kotwica, Niemiec, Kobzar, Bohdan & Wiatr (in 40 minutes)
[6] Lanti, Tran, Jocksch, Hariri, Brunner, Gheller & Villard (2016)
[7] Durand, Raffin & Faure (2012)
[8] Nakashima, Summura, Kikura & Miyake (2017)

Cell Index plus Offset

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

Position renormalized on the grid: $(x, y) \in [0; ncx) \times [0; ncy)$



$$\begin{cases} x = i\_x + dx \\ y = i\_y + dy \end{cases}$$

28 bytes per particle: `int` icell, `float` dx, dy, `double` vx, vy[9].

$i_{cell} \in \{0, 1, \ldots ncx \times ncy - 1\}$: one-to-one mapping with $(i\_x, i\_y) \in \{0, 1, \ldots ncx - 1\} \times \{0, 1, \ldots ncy - 1\}$, e.g.:

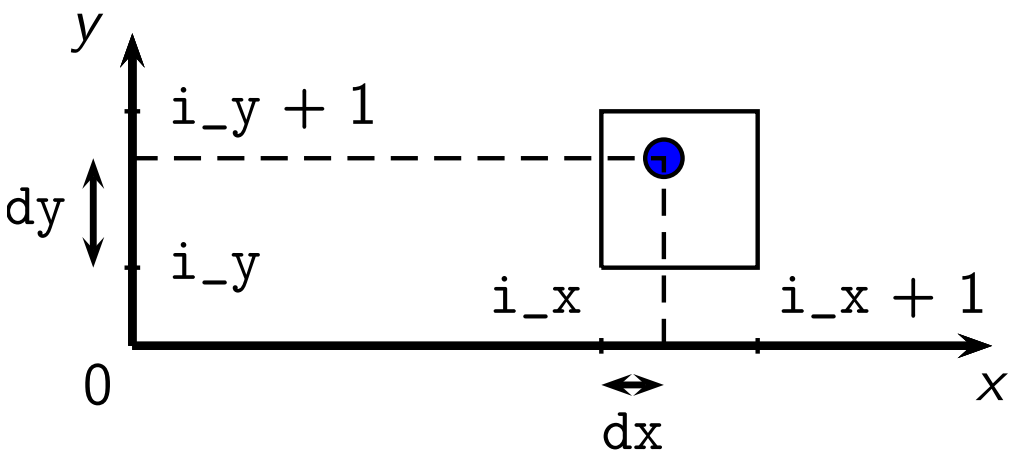
- $i_{cell} = i\_x \times ncy + i\_y$
- $\begin{cases} i\_x = i_{cell} / ncy \\ i\_y = modulo(i_{cell}, ncy) \end{cases}$

[9] Bowers, Albright, Yin, Bergen & Kwan (2008)

~~Cell Index plus~~ Offset

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

Position renormalized on the grid: $(x, y) \in [0; ncx) \times [0; ncy)$



$$\begin{cases} x = i\_x + dx \\ y = i\_y + dy \end{cases}$$

24 bytes per particle: ~~`int icell`~~, `float` dx, dy, `double` vx, vy[9].

First improvement of our method, shared with previous works that sort particles at each iteration: 14% memory saved.

Additional memory gains from our method: avoid allocating too many holes in the data structure.

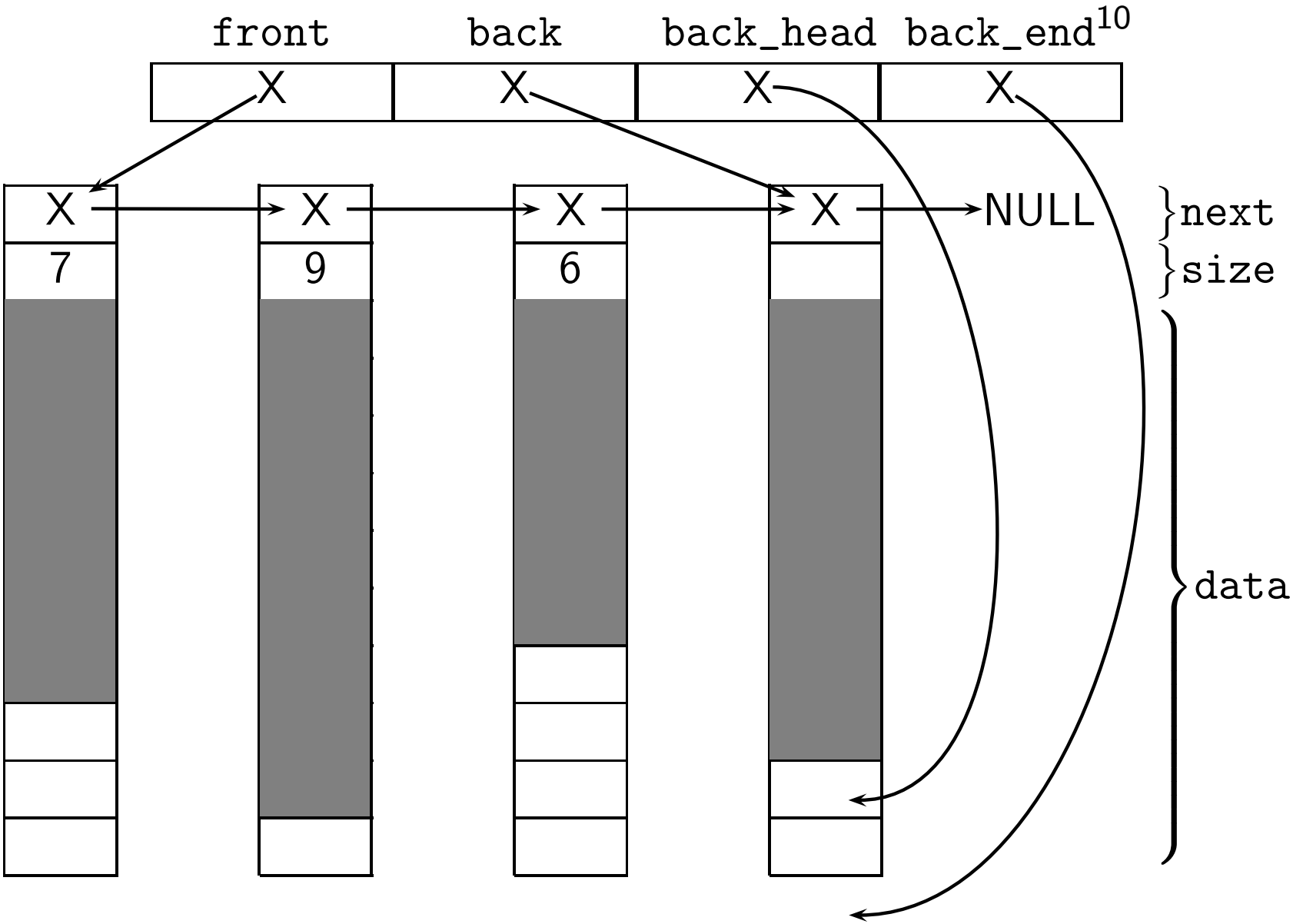
[9] Bowers, Albright, Yin, Bergen & Kwan (2008)

Goals for an Efficient Data Structure

In addition to keep the particles sorted at all times, we want:

- robustness: static arrays cannot simulate test cases in which a cell contains more particles than the statically chosen size
- cache efficiency: linked lists cause too many memory indirections
- no hidden constants: vectors (resizable arrays) incur a factor 2 overhead because of the resize operations
- multicore efficiency (1): as little atomic operations as possible
- multicore efficiency (2): avoid global refactoring of the data structure

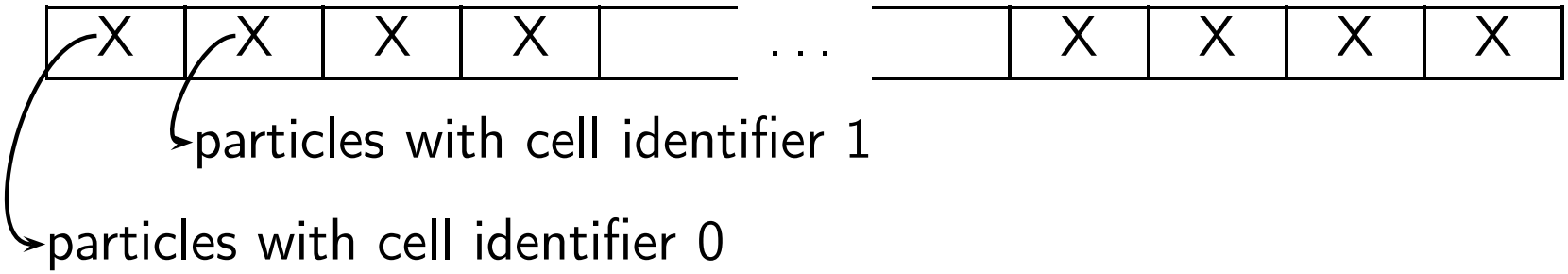
Chunk Bags: An Example



[10] Hanson (1990)

Chunk Bags: Particle Arrays

`chunkbag particles[nbCells]`

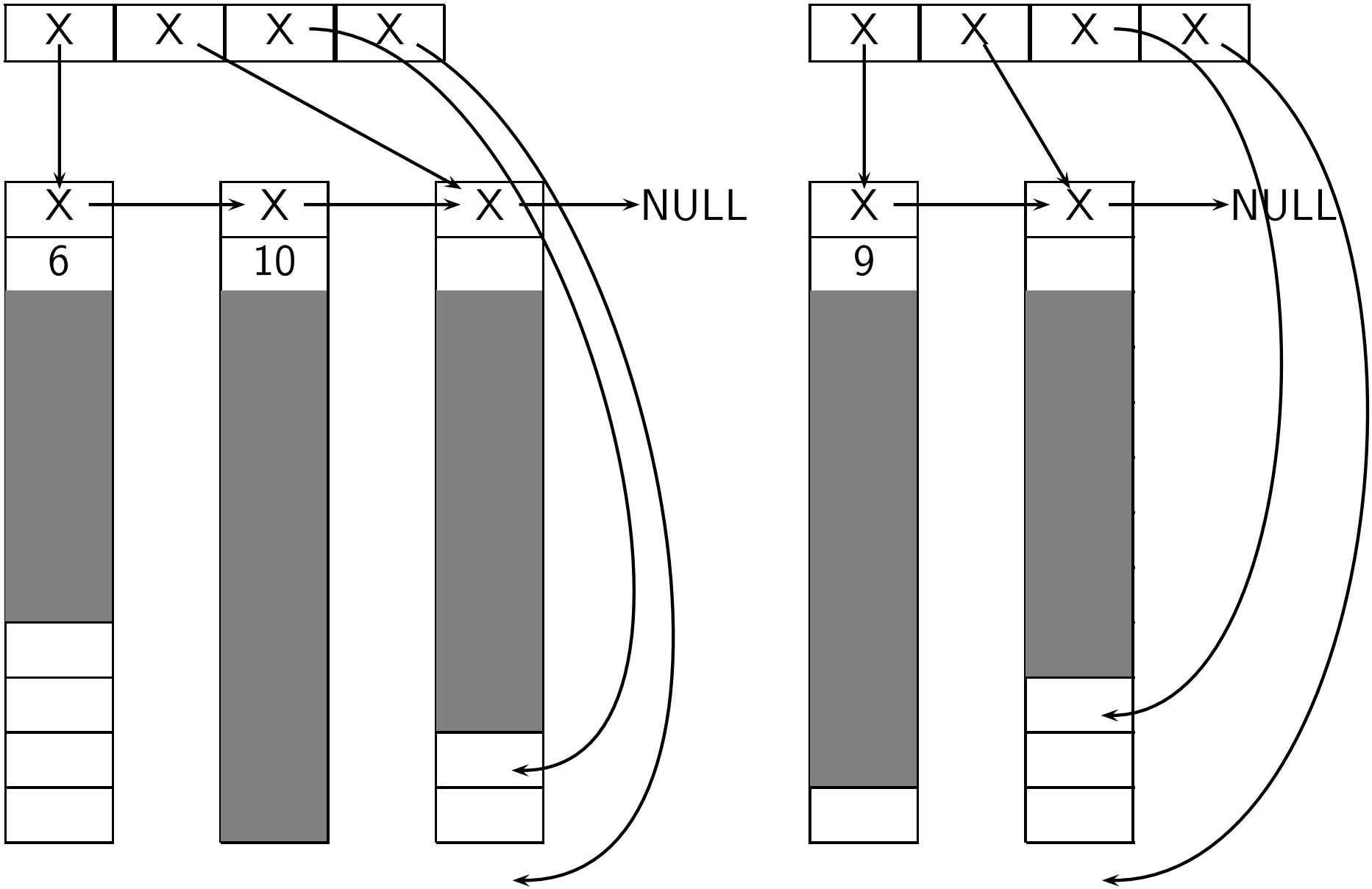


`chunkbag particlesNext[nbCores][nbCells]`

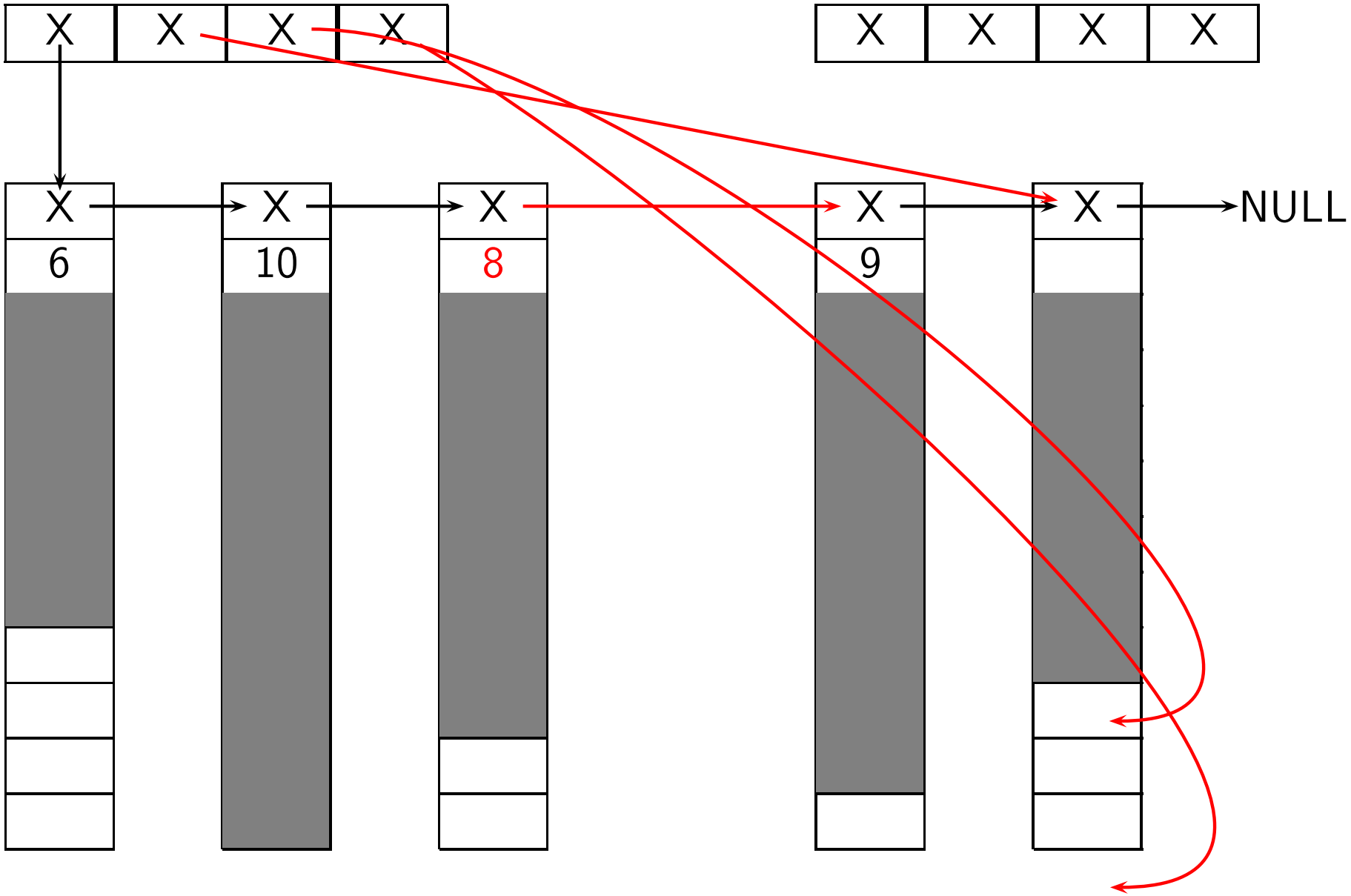
`particlesNext[i][j]`: the particles that the core *i* has treated and that have now, after update, the cell identifier *j*.

- race conditions are avoided
- `particlesNext[0][j]`, `particlesNext[1][j]`... `particlesNext[nbCores - 1][j]` need to be merged for the next iteration (into `particles[j]`)

Chunk Bags: Merge Operation



Chunk Bags: Merge Operation



Hybrid Parallel and Vectorized Code

- Among MPI processes: particle decomposition (each process keeps a constant subset of the particles and the whole grid)
- Advantage: computations are automatically balanced
- Drawback: communication overhead (`MPI_ALLREDUCE`) when using more than a few thousand MPI processors

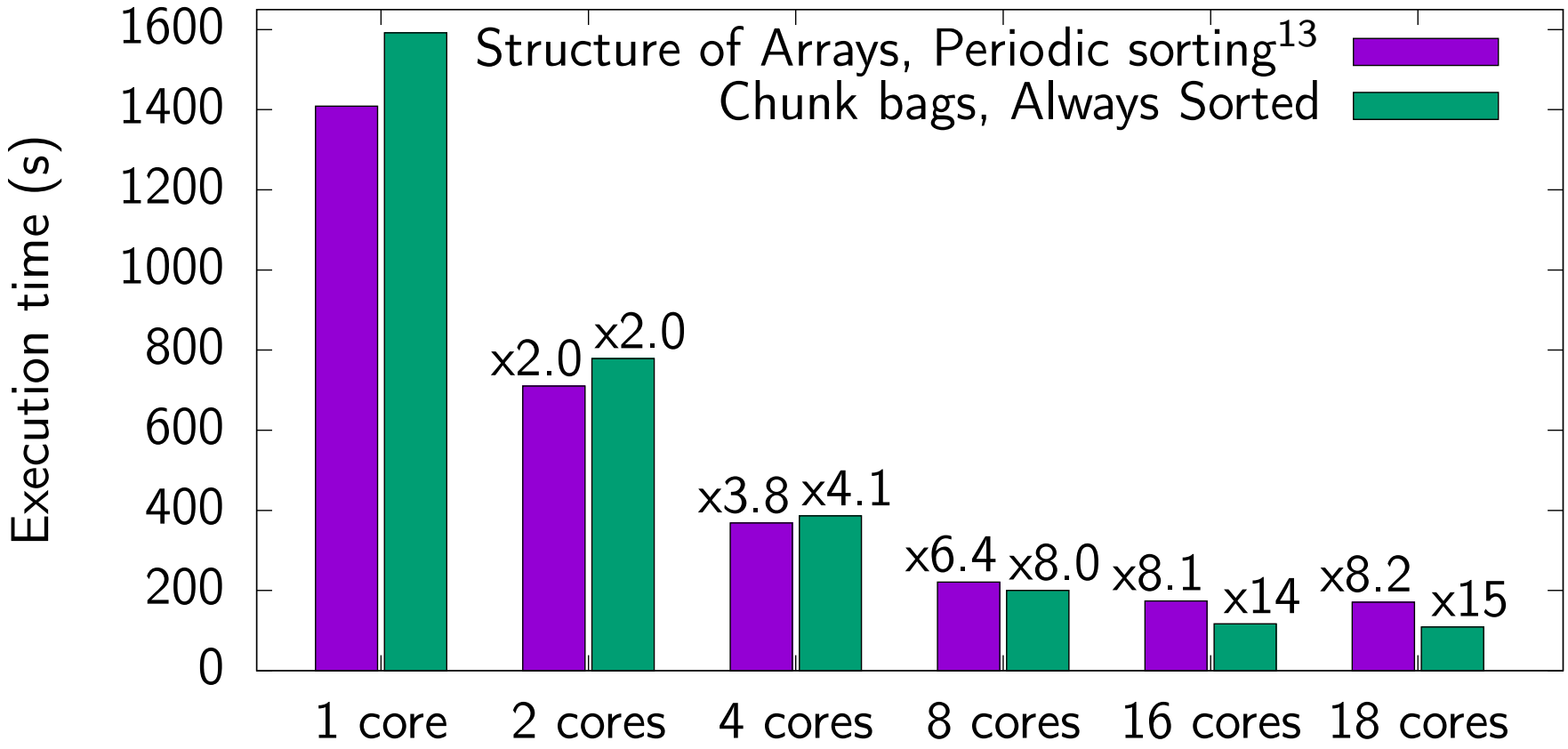
- On one socket: OpenMP pragmas
  `#pragma omp for` on the particle loop
  `reduction(+:rho[0:ncx*ncy][0:4])`[11] for accumulation

- On one core: automatic vectorization by `icc` and `gcc`
  loop fission for update-velocities
  vectorization over the corners for accumulation[12]

[11] Since OpenMP 4.5 in `C`.

[12] Vincenti, Lobet, Lehe, Sasanka & Vay (2016)

Strong Scaling on 18 Cores & 4 Memory Channels

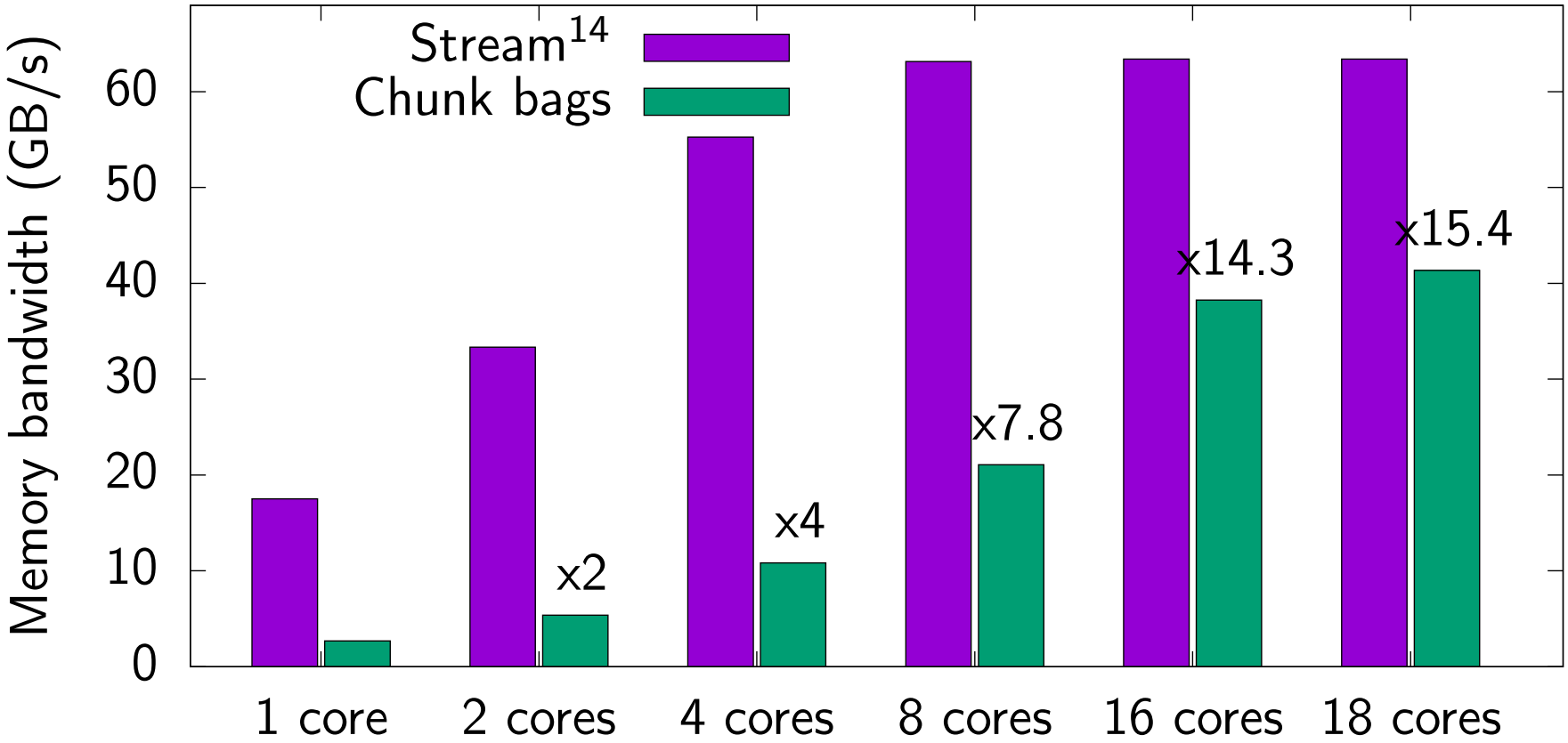


13% slower on single core, 36% faster on 18 cores.

128x128 grid, $900 \cdot 10^6$ particles, 100 iterations. Intel Broadwell (2016).

[13] Barsamian, Hirstoaga & Violard (2017)

Memory Bandwidth on 18 Cores & 4 Memory Channels

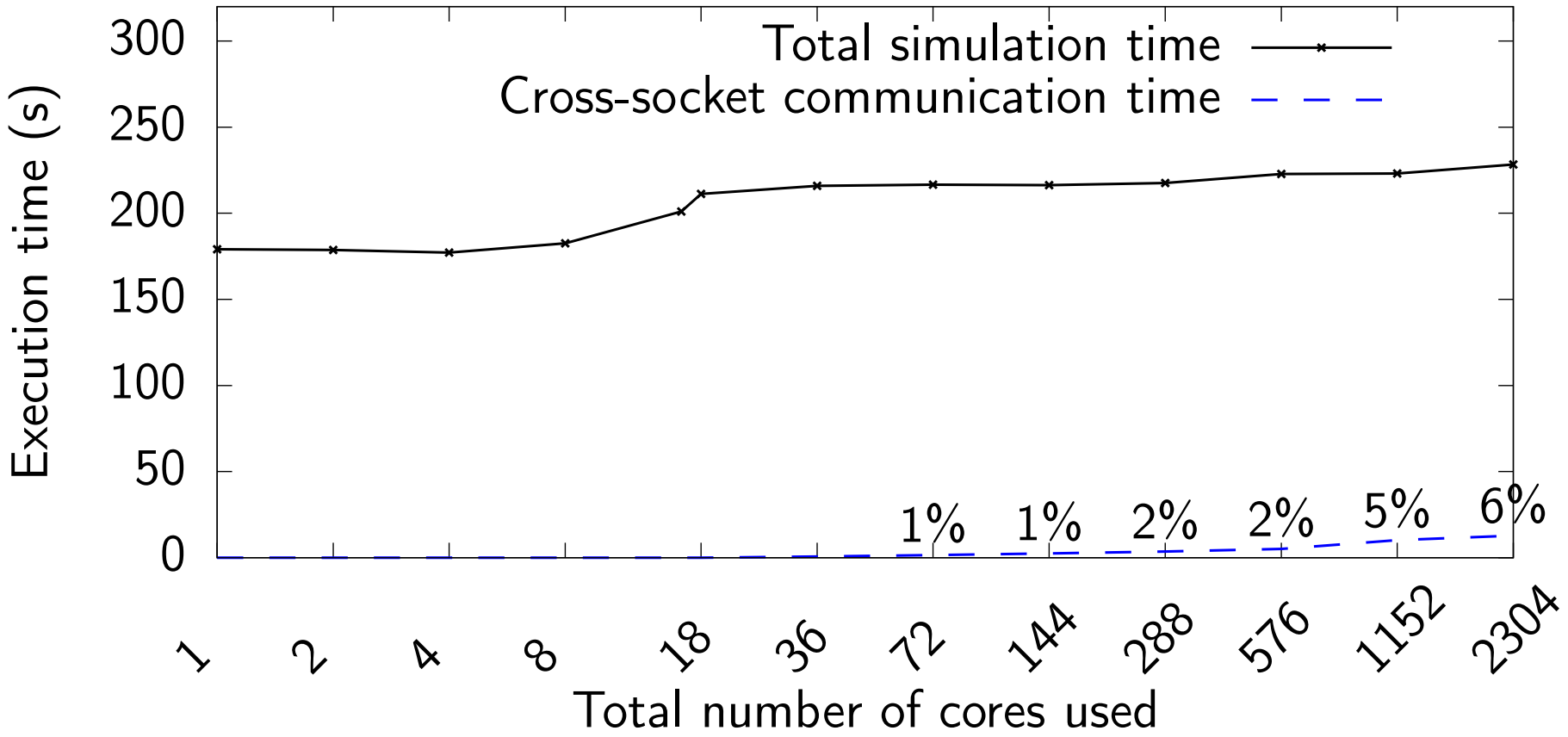


With $1.8 \cdot 10^9$ particles: 65% of the reachable peak given by the Stream test (theoretical peak: 76.8 GB/s). Bandwidth formula: nbIterations $\times$ nbParticles $\times$ sizeof(particle) $\times$ 2/executionTime.

128x128 grid, $100 \cdot 10^6$ particles / core, 100 iterations. Intel Broadwell (2016).

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

Weak Scaling on 2,304 Cores



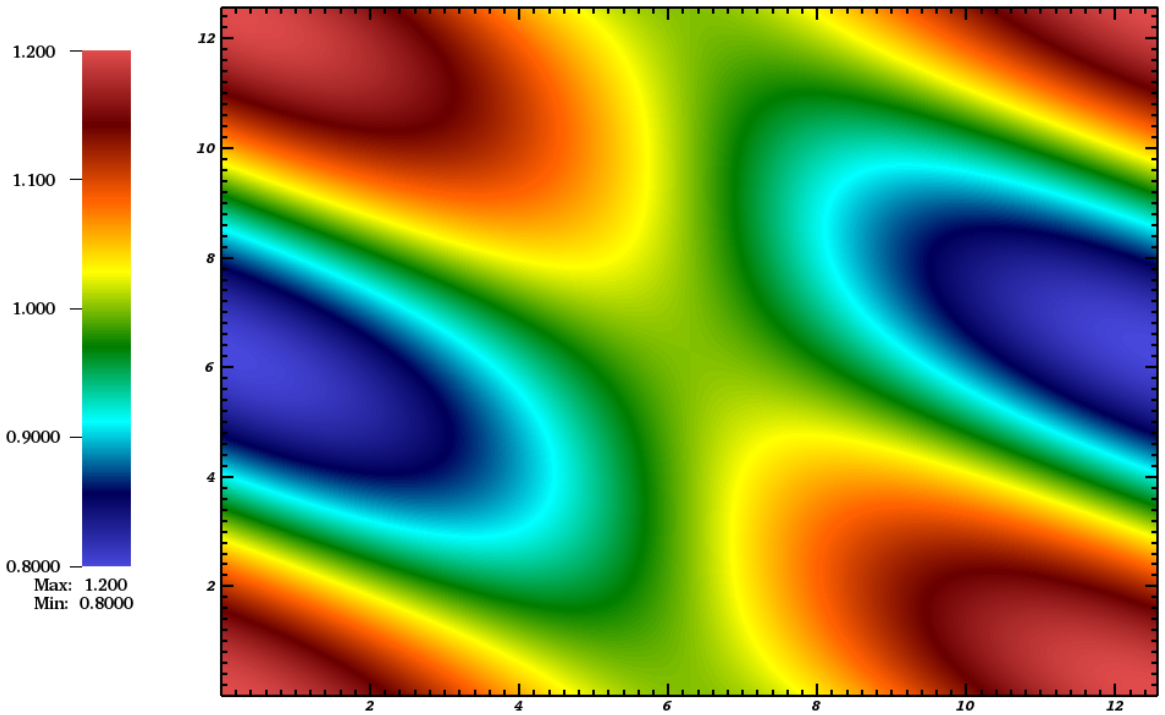
92% parallel efficiency on 2,304 cores (230 billion particles)

128x128 grid, $100 \cdot 10^6$ particles / core, 100 iterations. Intel Broadwell (2016).

Example Simulation: Two-stream instability

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.05$, initial condition[15]:

$$f(\vec{x}, \vec{v}, 0) = \left(1 + 0.1\left(\cos\left(\frac{y}{2}\right) + \cos\left(\frac{x+y}{2}\right)\right)\right)\frac{v_x^2}{2\pi}e^{-\frac{v_x^2+v_y^2}{2}}$$



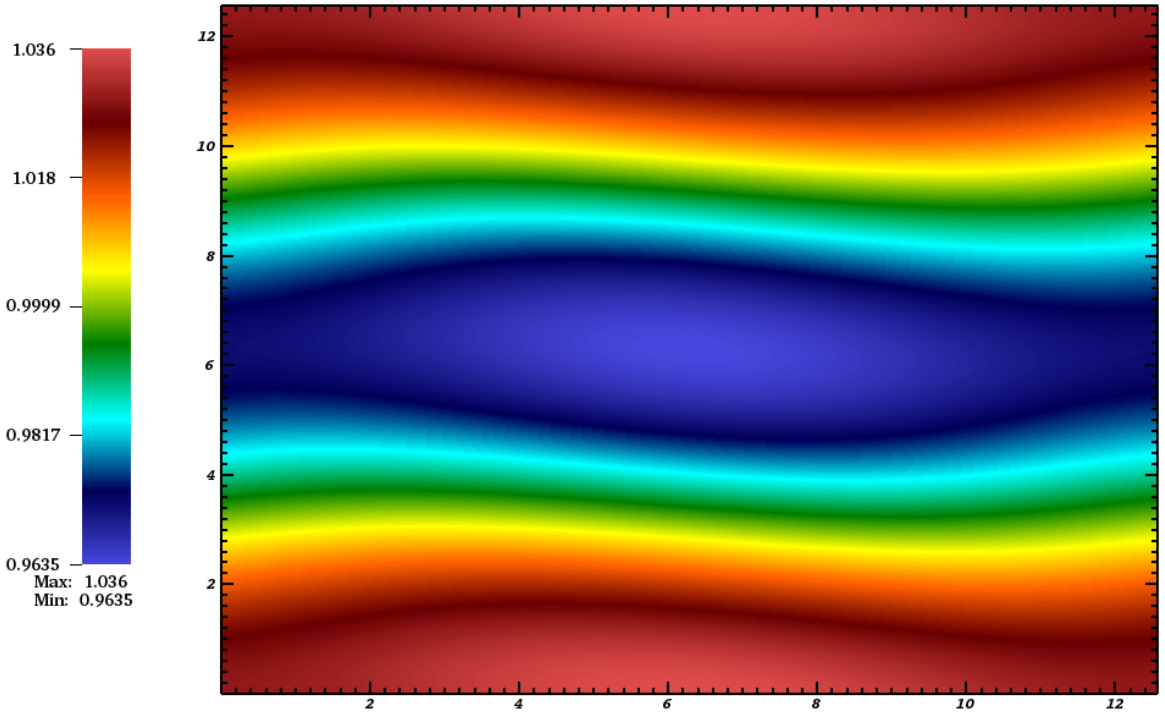
Iteration 0

[15] Barsamian, Bernier, Hirstoaga & Mehrenberger, 2017

Example Simulation: Two-stream instability

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.05$, initial condition[15]:

$$f(\vec{x}, \vec{v}, 0) = \left(1 + 0.1\left(\cos\left(\frac{y}{2}\right) + \cos\left(\frac{x+y}{2}\right)\right)\right)\frac{v_x^2}{2\pi}e^{-\frac{v_x^2+v_y^2}{2}}$$



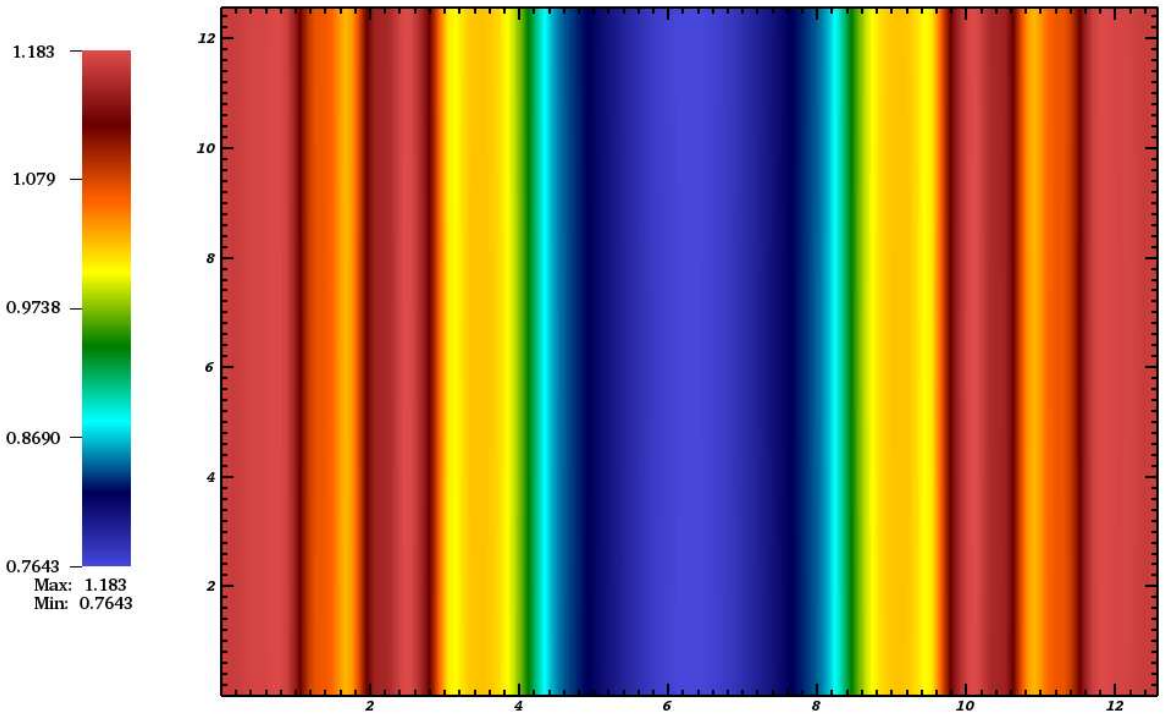
Iteration 100

[15] Barsamian, Bernier, Hirstoaga & Mehrenberger, 2017

Example Simulation: Two-stream instability

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.05$, initial condition[15]:

$$f(\vec{x}, \vec{v}, 0) = \left(1 + 0.1\left(\cos\left(\frac{y}{2}\right) + \cos\left(\frac{x+y}{2}\right)\right)\right)\frac{v_x^2}{2\pi}e^{-\frac{v_x^2+v_y^2}{2}}$$



Iteration 1,000

[15] Barsamian, Bernier, Hirstoaga & Mehrenberger, 2017

Conclusions

- contributions
  - introduce chunk bags to keep particles sorted at all time without significant memory overhead
  - design an algorithm where each particle is loaded/written exactly once from/to main memory per iteration
  - and where particles are processed using state-of-the-art vectorization techniques, with efficient OpenMP load balancing
- on each socket (18 cores & 4 memory channels), near-optimal memory consumption and bandwith usage and processing time
  - 861 million particles / second, or 48 million / second / core
  - 65% of the maximum bandwidth
- 92% parallel efficiency on 2,304 cores with 230 billion particles

ybarsamian@unistra.fr