


Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell (PIC) Code

Yann Barsamian, Sever Hirstoaga, Éric Violard

Université de Strasbourg

CNRS I³ Laboratory (ICPS Team)
INRIA (CAMUS Team)

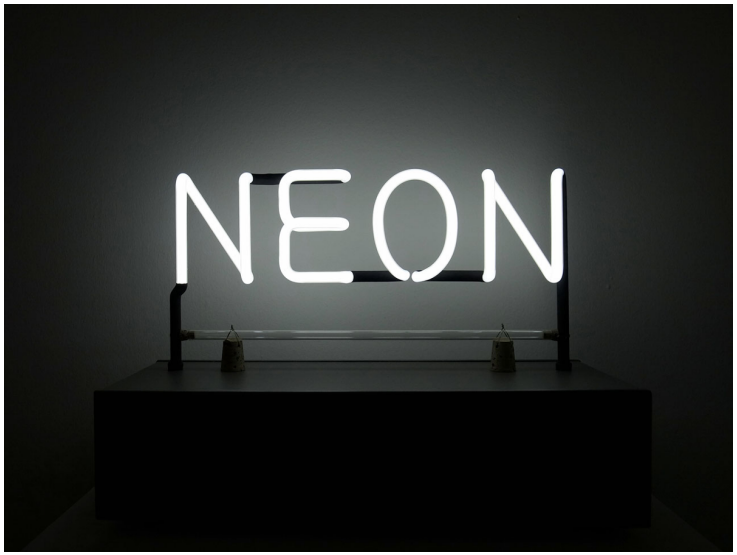
PDSEC 2017, Orlando (Florida, U.S.A.)
June 2017

Outline

- What is plasma useful for? (physics)
- How can we model its dynamics? (mathematics)
- What can we optimize on one core?
- How does the code scale?

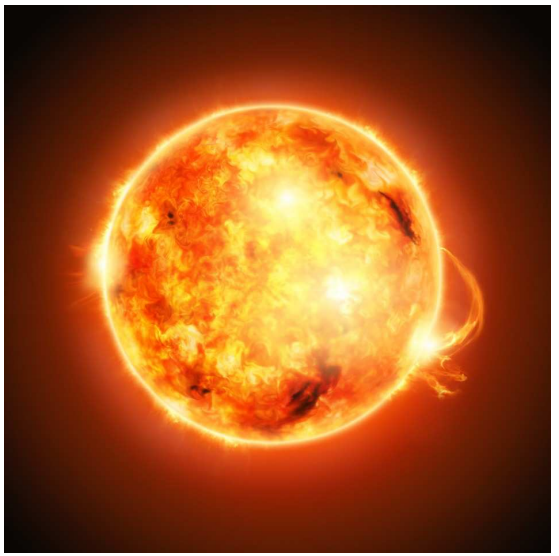
Plasma: The Fourth State of Matter

Lightning



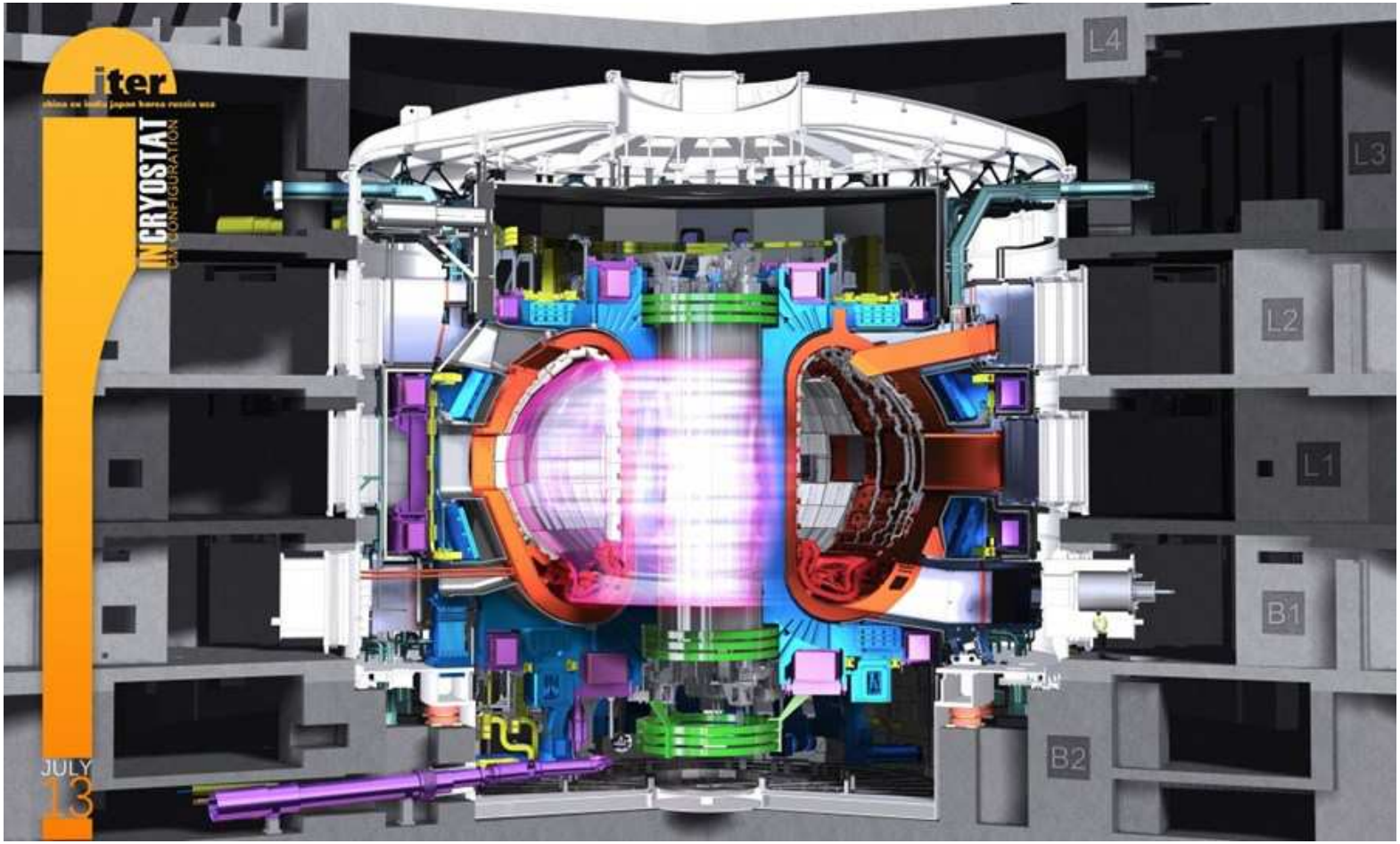
Fluorescent Light

Plasma Etching



Sun

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 (one species, dimensionless quantities)

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

- $f(\vec{x}, \vec{v}, t)$: distribution function of the electrons
- $\vec{E}(\vec{x}, t)$: the self-induced electric field
- t: time
- $\vec{x}$: position (2d with periodic boundaries)
- $\vec{v}$: velocity (2d)
- $\rho(\vec{x}, t) = 1 - \int f(\vec{x}, \vec{v}, t) d\vec{v}$: volume charge density

Kinetic Modeling (one species, dimensionless quantities)

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f - \vec{E} \cdot \nabla_{\vec{v}} f = 0 \qquad \text{Vlasov}$$

Kinetic Modeling (one species, dimensionless quantities)

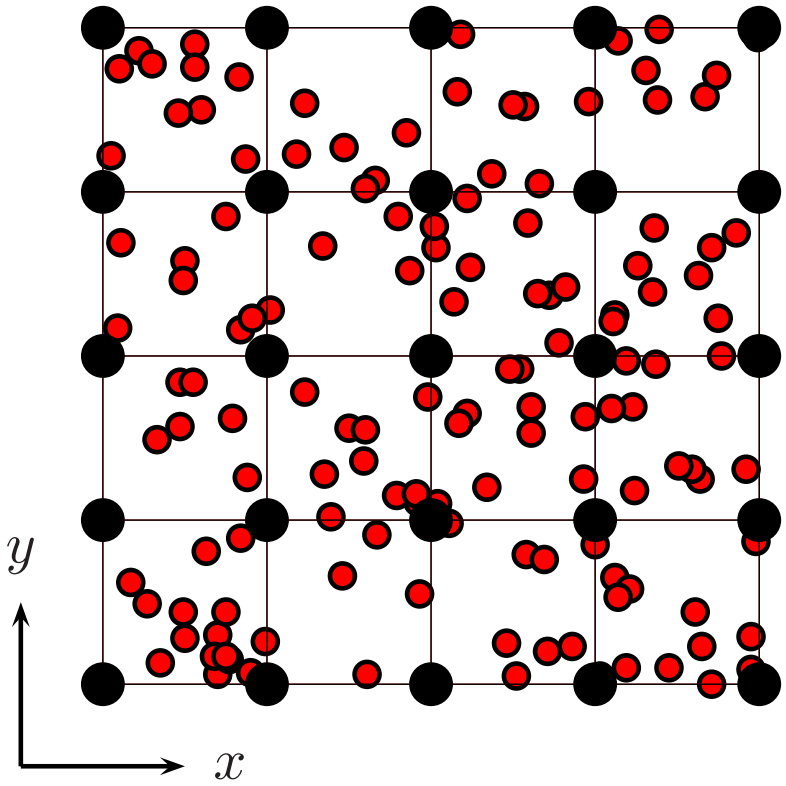
$$\frac{\partial f}{\partial t} + \overrightarrow{v} \cdot \nabla_{\overrightarrow{x}} f - \overrightarrow{E} \cdot \nabla_{\overrightarrow{v}} f = 0 \qquad \text{Vlasov}$$

Characteristics of this equation:

$$\begin{cases} \dfrac{\mathrm{d}x}{\mathrm{d}t} = v \\ \dfrac{\mathrm{d}v}{\mathrm{d}t} = -E \end{cases} \qquad \text{Newton's second law}$$

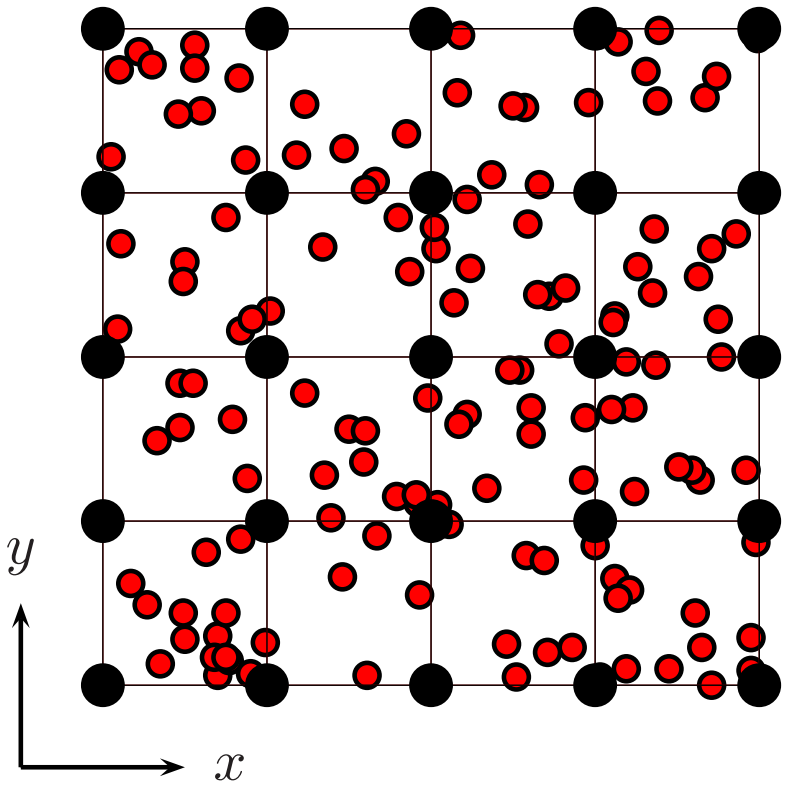
Particle-in-Cell Methods

- discretization of f via N numerical particles (red)
- discretization of $\overrightarrow{E}$ and ρ via $ncx \times ncy$ grids (black)



Particle-in-Cell Methods

- discretization of f via N numerical particles (red)
- discretization of $\overrightarrow{E}$ and ρ via $ncx \times ncy$ grids (black)

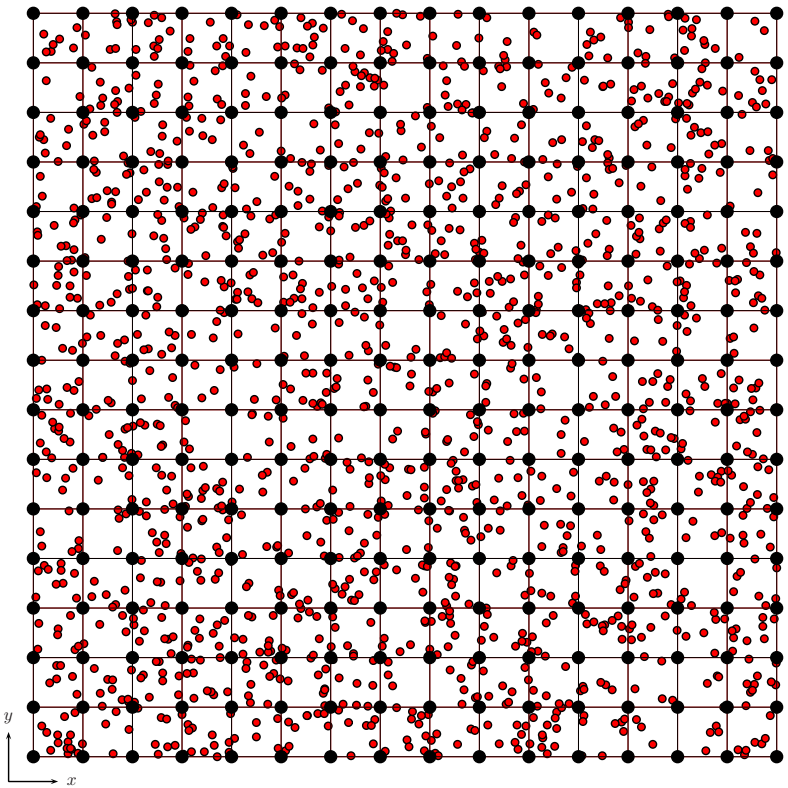


Issues:

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

Particle-in-Cell Methods

- discretization of f via N numerical particles (red)
- discretization of $\overrightarrow{E}$ and ρ via $ncx \times ncy$ grids (black)

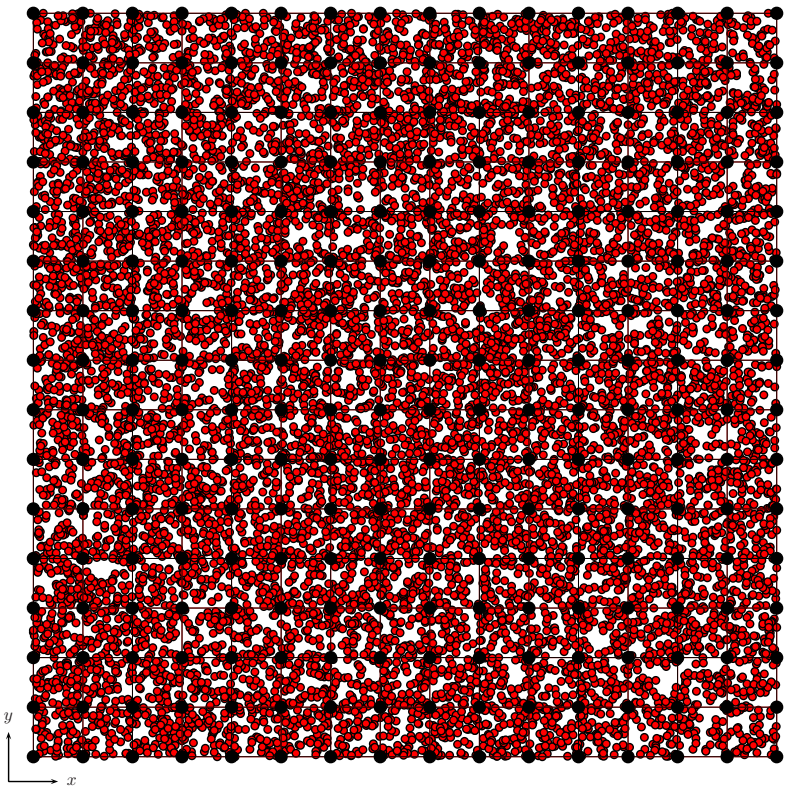


Issues:

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

Particle-in-Cell Methods

- discretization of f via N numerical particles (red)
- discretization of $\overrightarrow{E}$ and ρ via $ncx \times ncy$ grids (black)

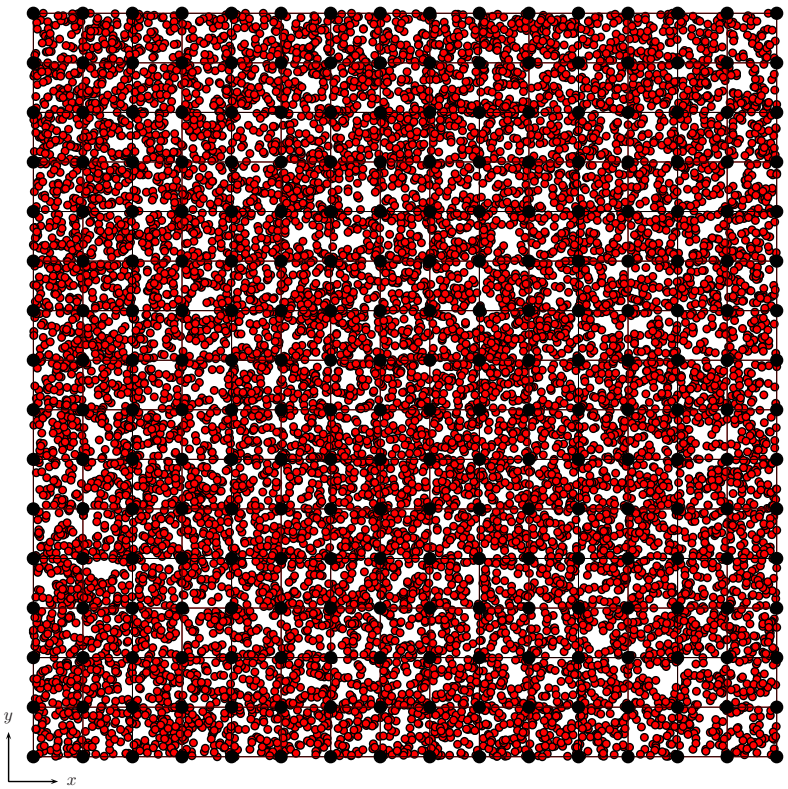


Issues:

- Physical effects on small 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

Particle-in-Cell Methods

- discretization of f via N numerical particles (red)
- discretization of $\overrightarrow{E}$ and ρ via $ncx \times ncy$ grids (black)



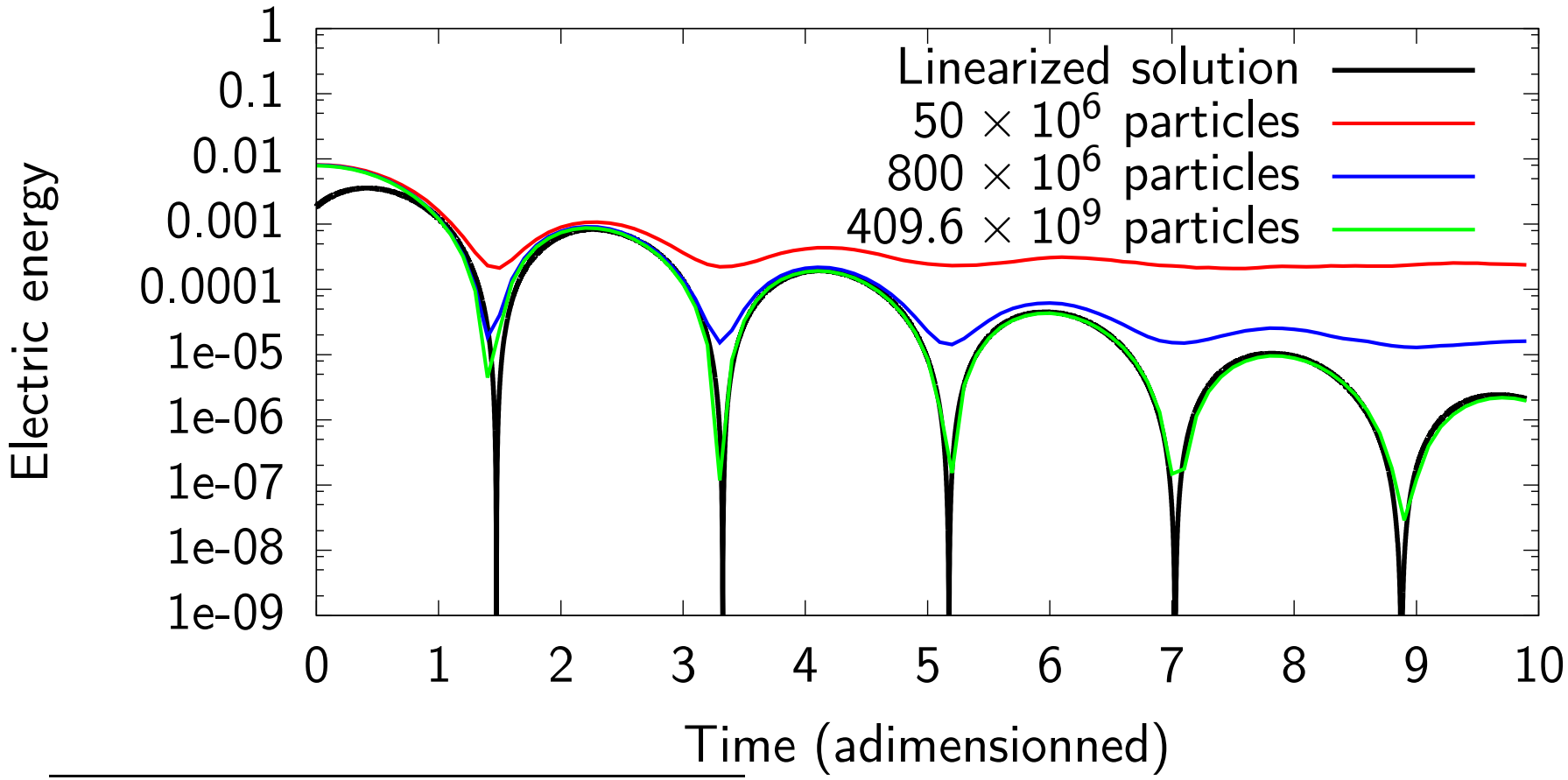
Issues:

- Physical effects on small 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
  $\Rightarrow$ efficient data structures

Code Verification: Ландау (Landau) Damping

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.1$, initial condition[3]:

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



[3] Ландау (1946)

Particle-in-Cell Pseudo-Code

Initialization:

1 Initialize N particles — `struct particle[N]`

2 Compute ρ and E (Ex and Ey) at $t = 0$ — `double[ncx][ncy]`

Algorithm:

```
3   Foreach time iteration do
4       If (condition) then
5           Sort the particles[4]                          O(N) counting sort
6       End If
7       Set all cells of ρ to 0
8       Foreach particle do
9           Update the velocity                            v+ = −EΔt
10          Update the position                            x+ = vΔt
11          Accumulate the charge on the nearest ρ cells
12      End Foreach
13      Compute E from ρ                                   FFT Poisson solver
14  End Foreach
```

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

Particle-in-Cell Pseudo-Code

Initialization:		
1	Initialize N particles	`struct particle[N]`
2	Compute ρ and E (Ex and Ey) at $t = 0$	`double[ncx][ncy]`
Algorithm:		
3	**Foreach** time iteration **do**	
4	**If** (*condition*) **then**	
5	Sort the particles[4]	15%
6	**End If**	
7	Set all cells of ρ to 0	
8	**Foreach** particle **do**	
9	Update the velocity	35%
10	Update the position	35%
11	Accumulate the charge on the nearest ρ cells	15%
12	**End Foreach**	
13	Compute E from ρ	<1%[5]
14	**End Foreach**	

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

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

Overall Optimization Gains

	T (s)	%	Acc. %
Baseline [6]	120.4	0.0	0.0
+ Loop Hoisting	113.4	5.8	5.8
+ Loop Fission	97.9	13.7	18.7
+ Redondant arrays (E, ρ)[7,8]	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 (2013).

[6] Chacon-Golcher, Hirstoaga & Lutz (2016), http://selalib.gforge.inria.fr/

[7] Bowers & Li (2003)

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

Overall Optimization Gains

	T (s)	%	Acc. %
Baseline [6]	120.4	0.0	0.0
+ Loop Hoisting	113.4	5.8	5.8
+ Loop Fission	97.9	13.7	18.7
+ Redondant arrays (E, ρ)[7,8]	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 (2013).

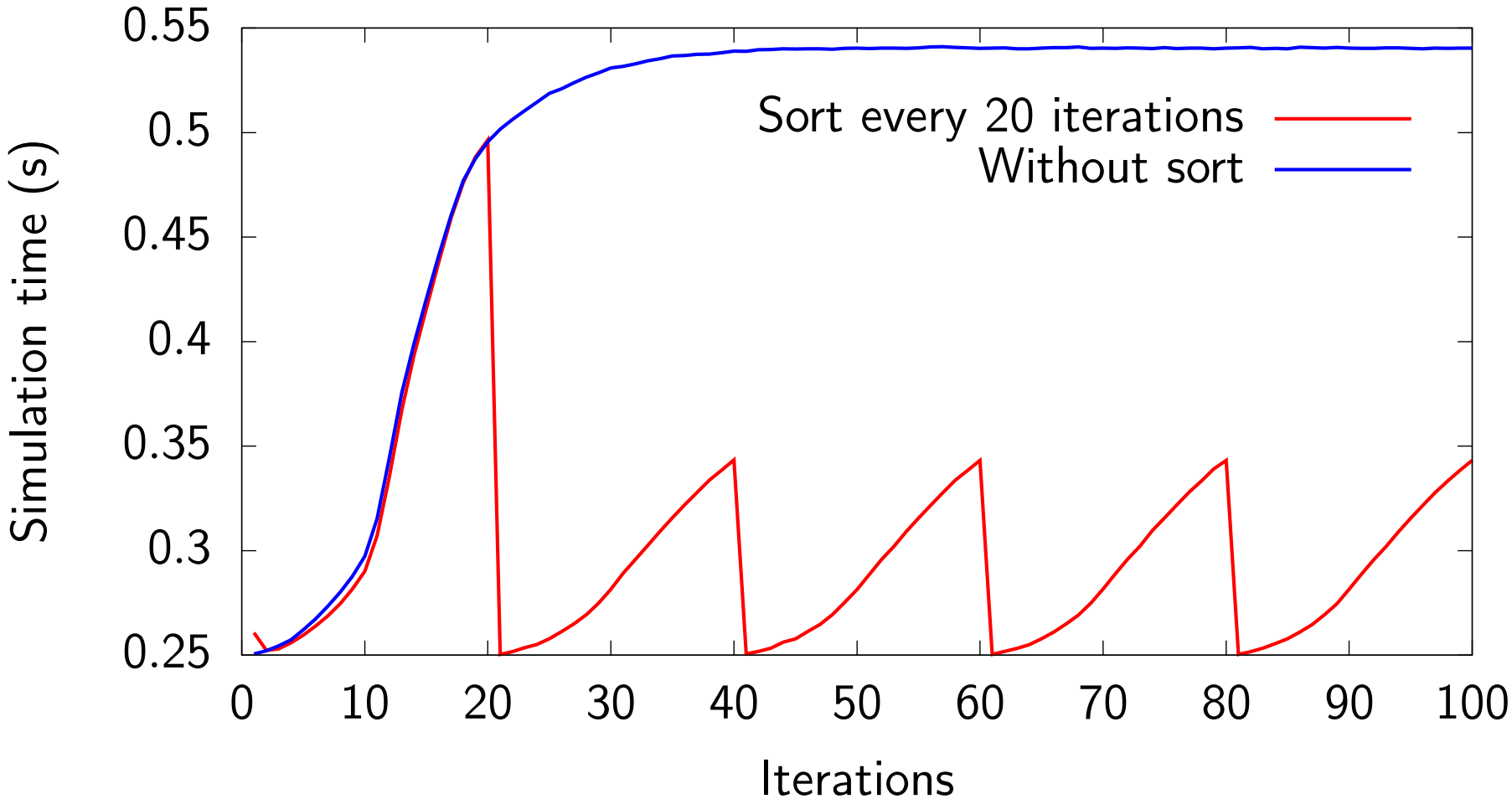
75 million particles processed/second on one core.

[6] Chacon-Golcher, Hirstoaga & Lutz (2016), http://selalib.gforge.inria.fr/

[7] Bowers & Li (2003)

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

Sorting of the Particle Array

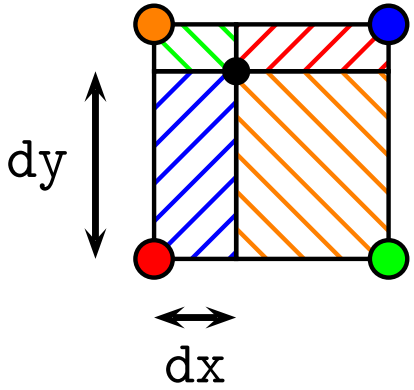


CPU time for 50,000,000 particles, with a 128×128 mesh, $\Delta t = 0.1$. Without sort: 87 s ; with sort: 67 s (including 8.6 s of sorting not shown here). Architecture: Intel Haswell (2013).

Particle-Mesh Interaction: Linear Interpolation

The Cloud-in-Cell[9] method:

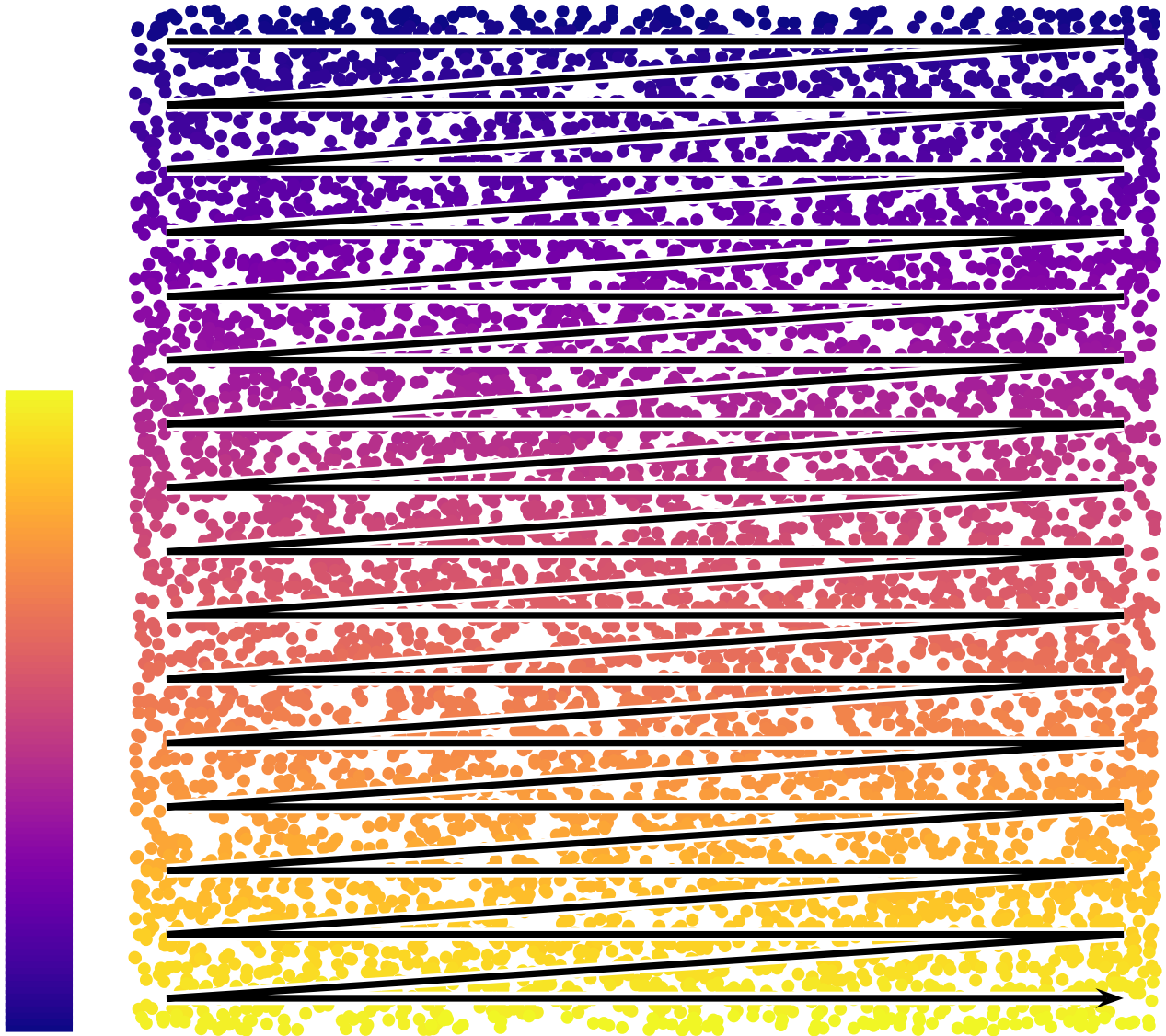
$$v+ = -E\Delta t$$




```
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);
}
```

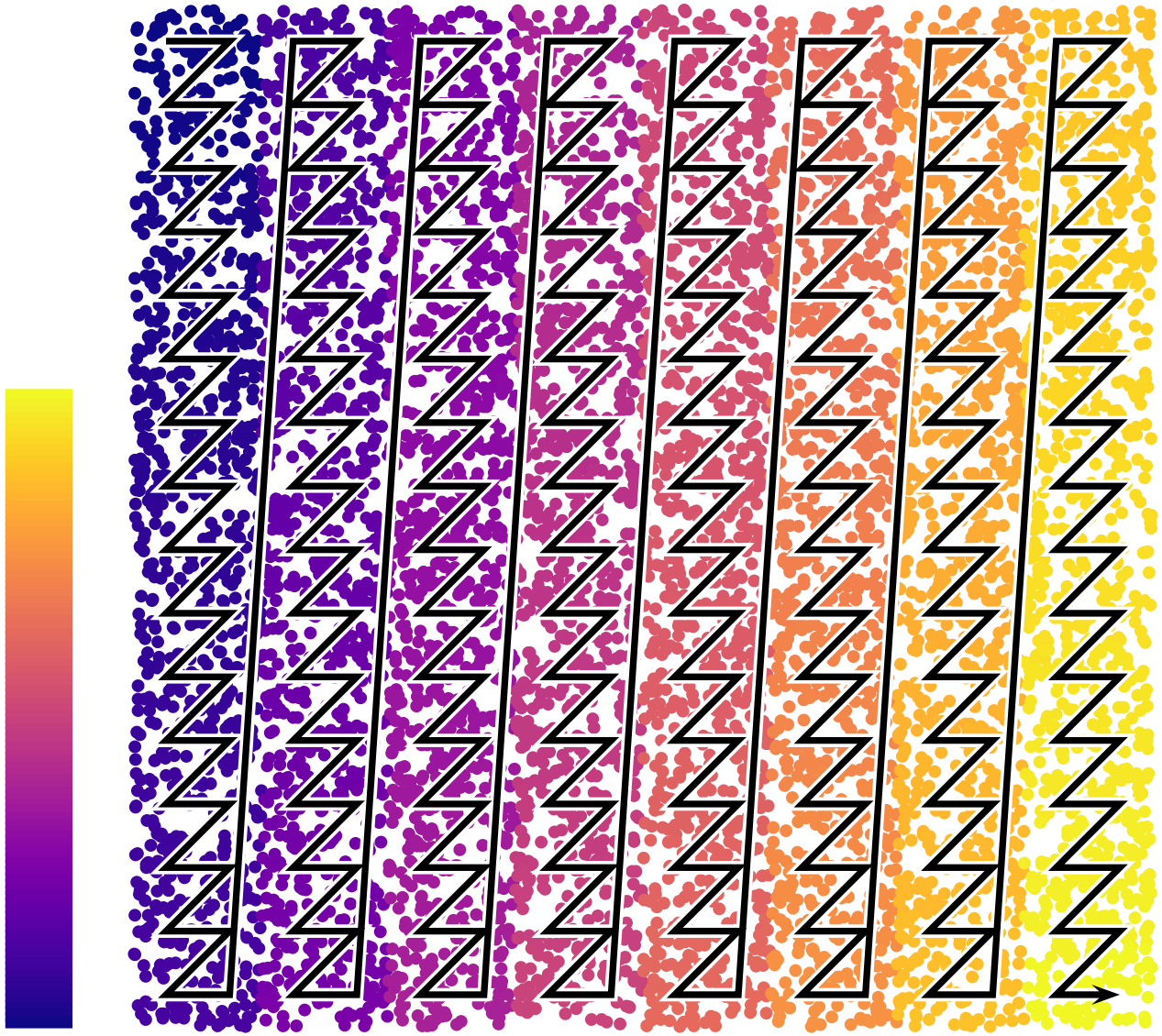
[9] Birdsall & Fuss (1969)

Space-Filling Curves for E and ρ: Drawings



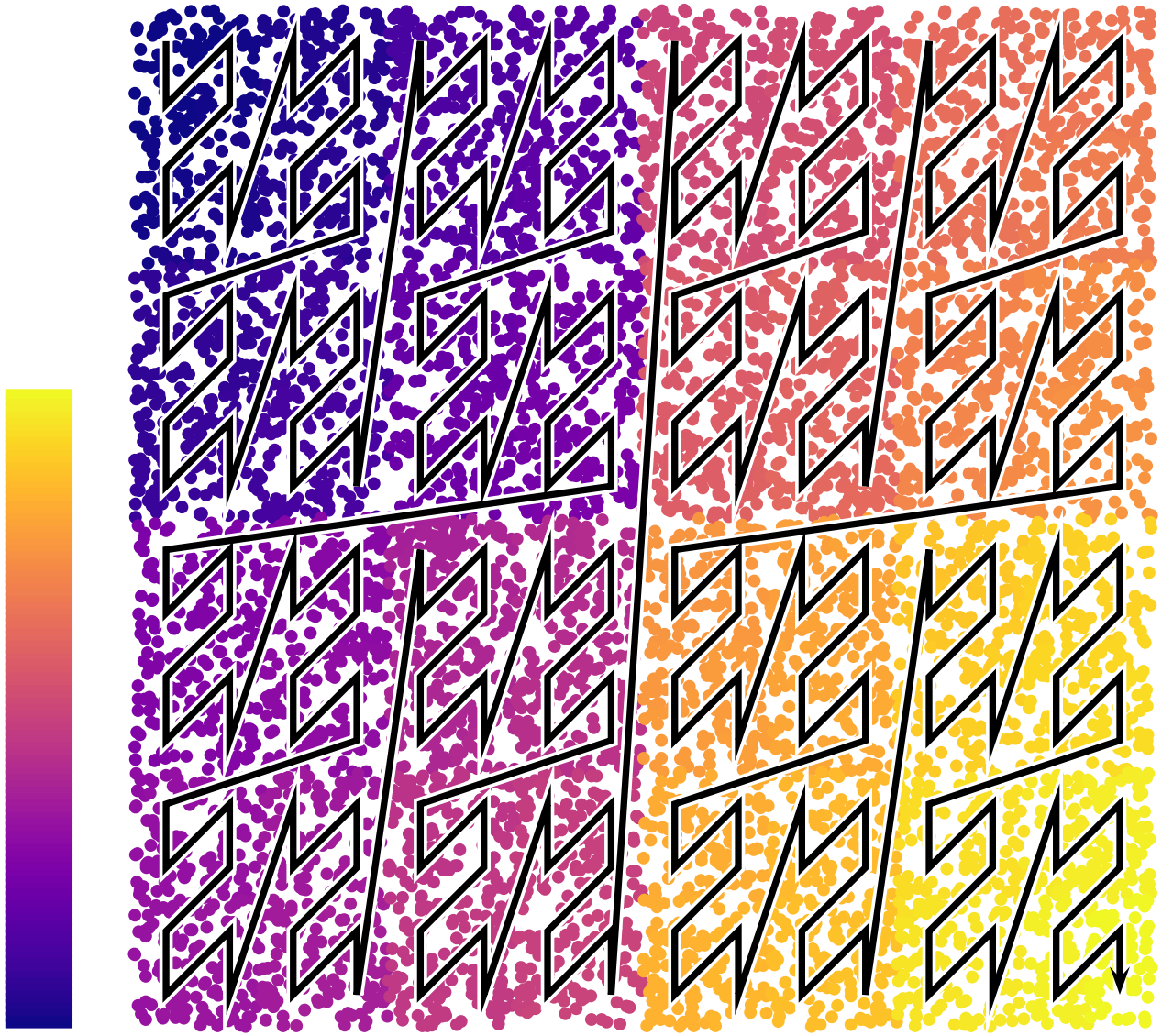
Row-Major (Standard layout in C)

Space-Filling Curves for E and ρ: Drawings



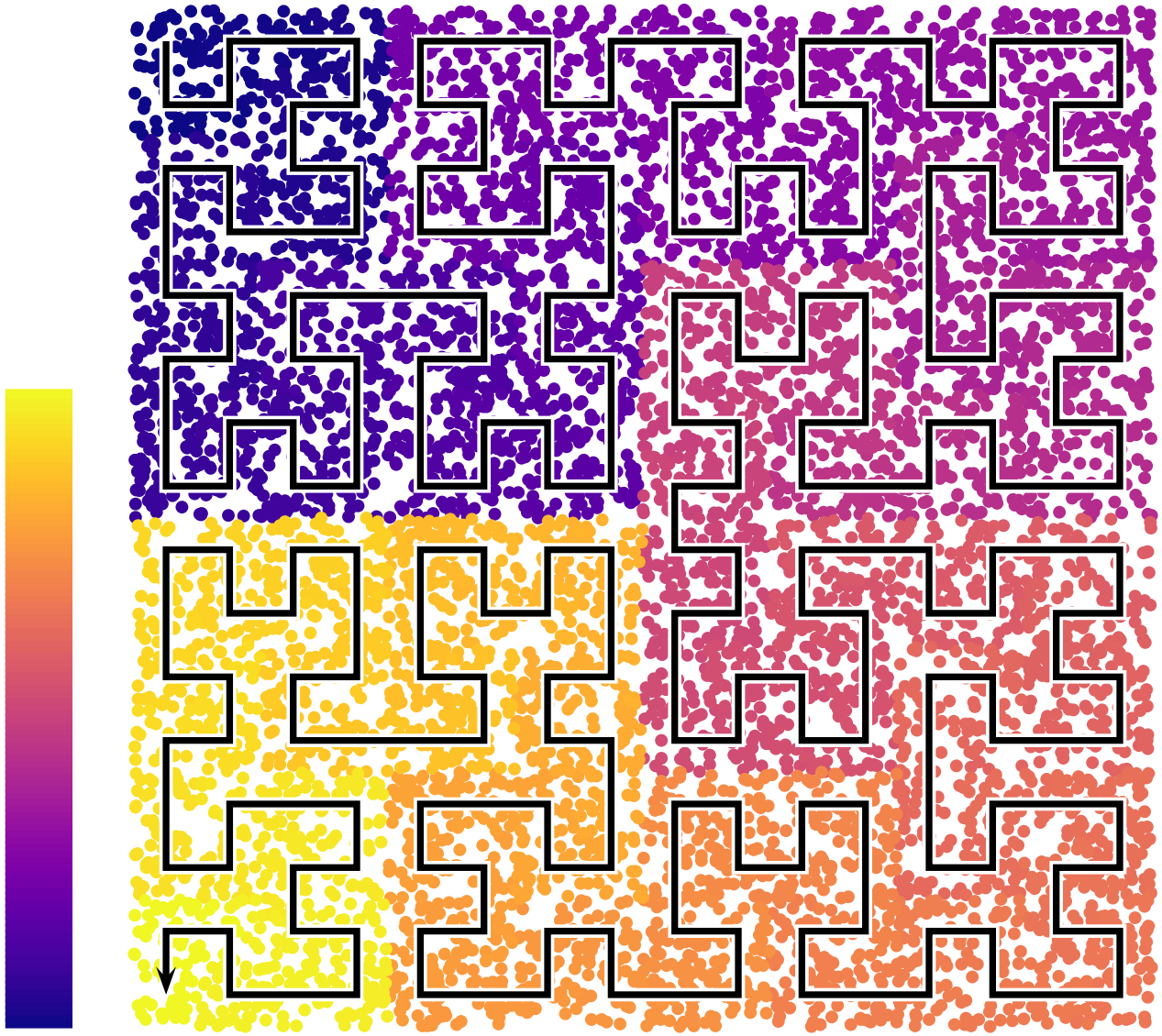
L4D curve (Chatterjee, Jain, Lebeck, Mundhra, Thottethodi, 1999)

Space-Filling Curves for E and ρ: Drawings



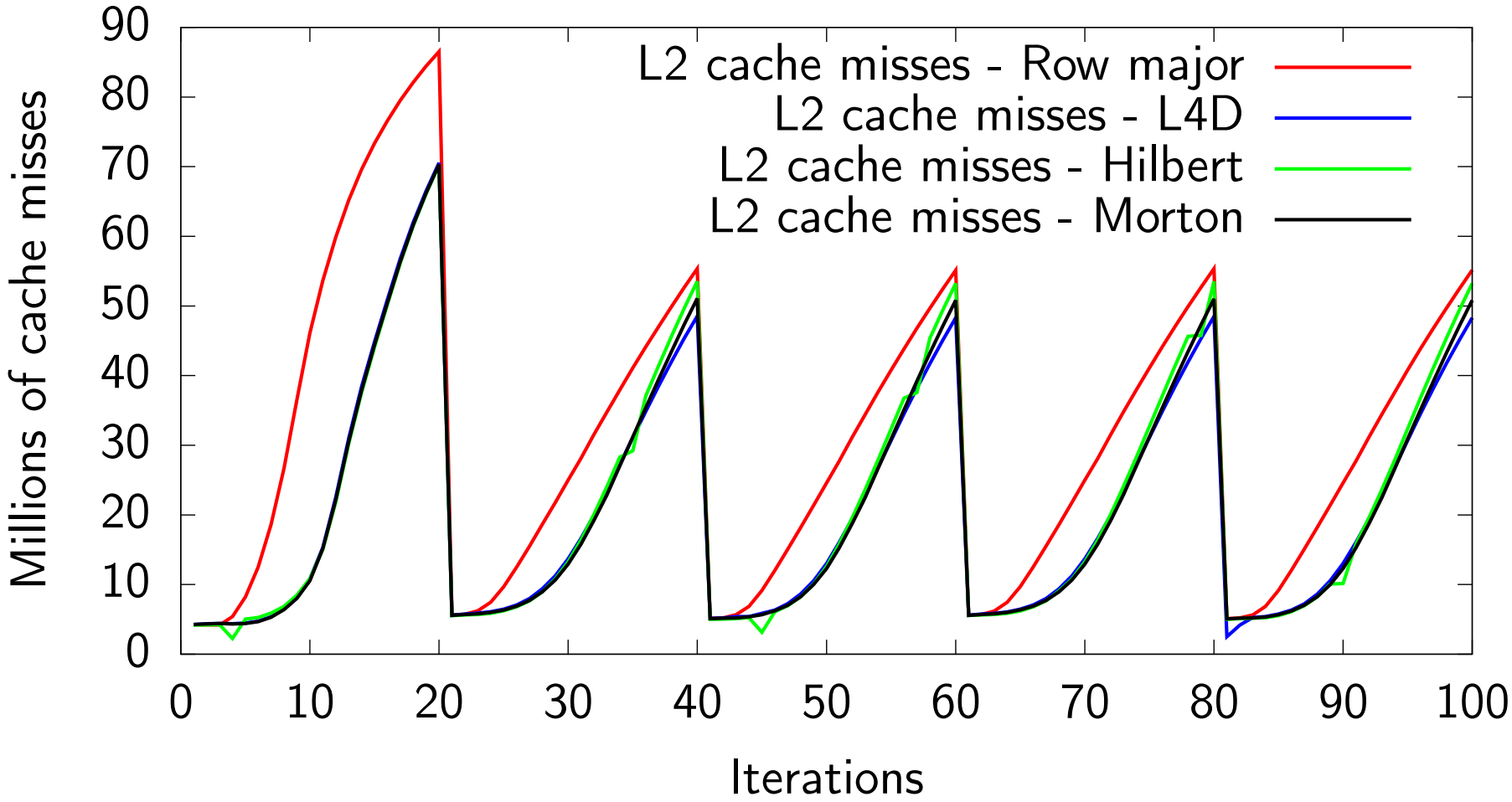
Morton curve (Morton, 1966)

Space-Filling Curves for E and ρ: Drawings



Hilbert curve (Hilbert, 1891)

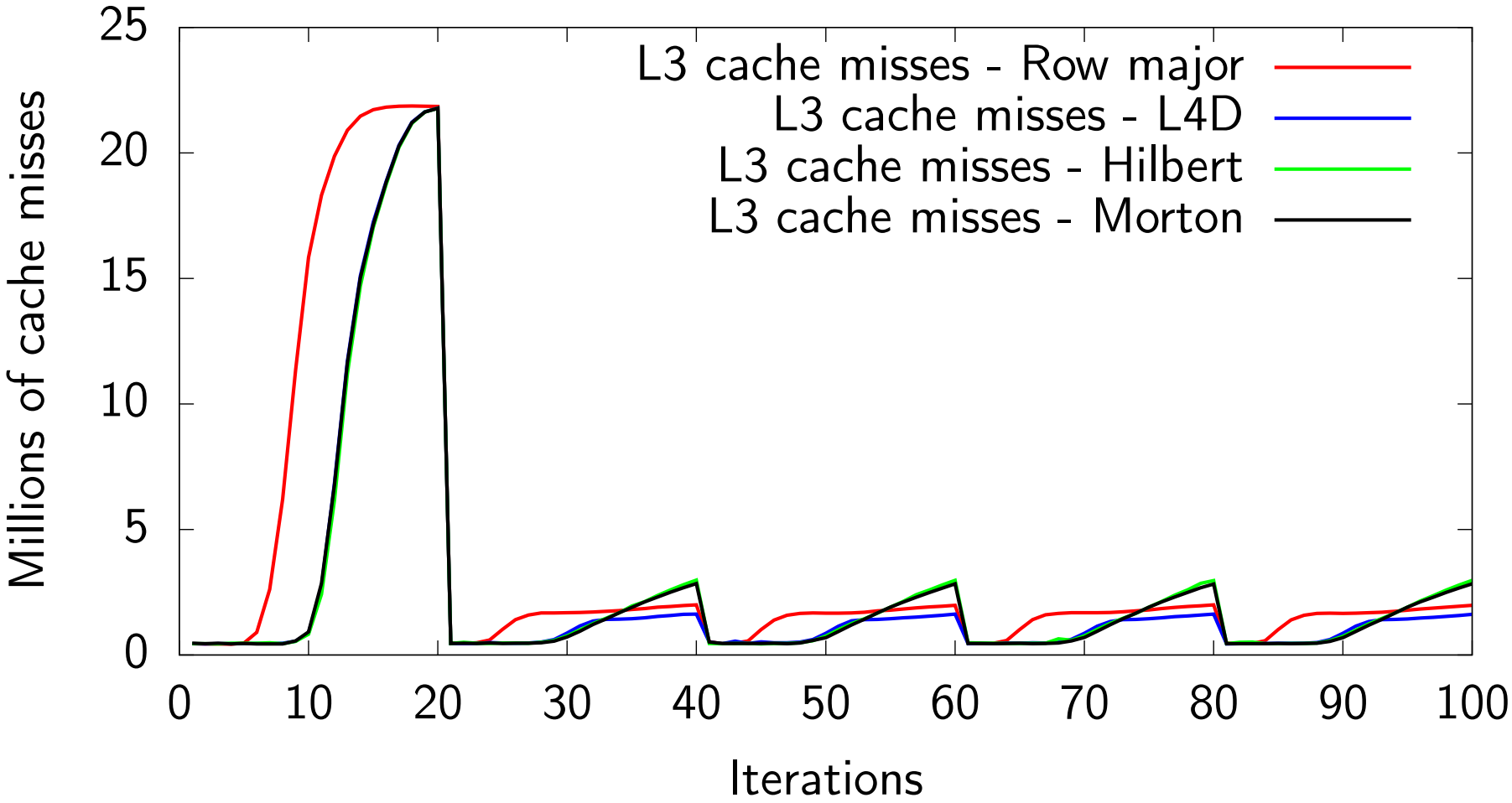
Space-Filling Curves for E and ρ: Cache Misses Results



Number of Level 2 cache misses[10], for 50,000,000 particles, with a 128×128 mesh, $\Delta t = 0.1$. Architecture: Intel Haswell (2013).

[10] Perf. Application Prgm. Interface (PAPI): http://icl.cs.utk.edu/papi

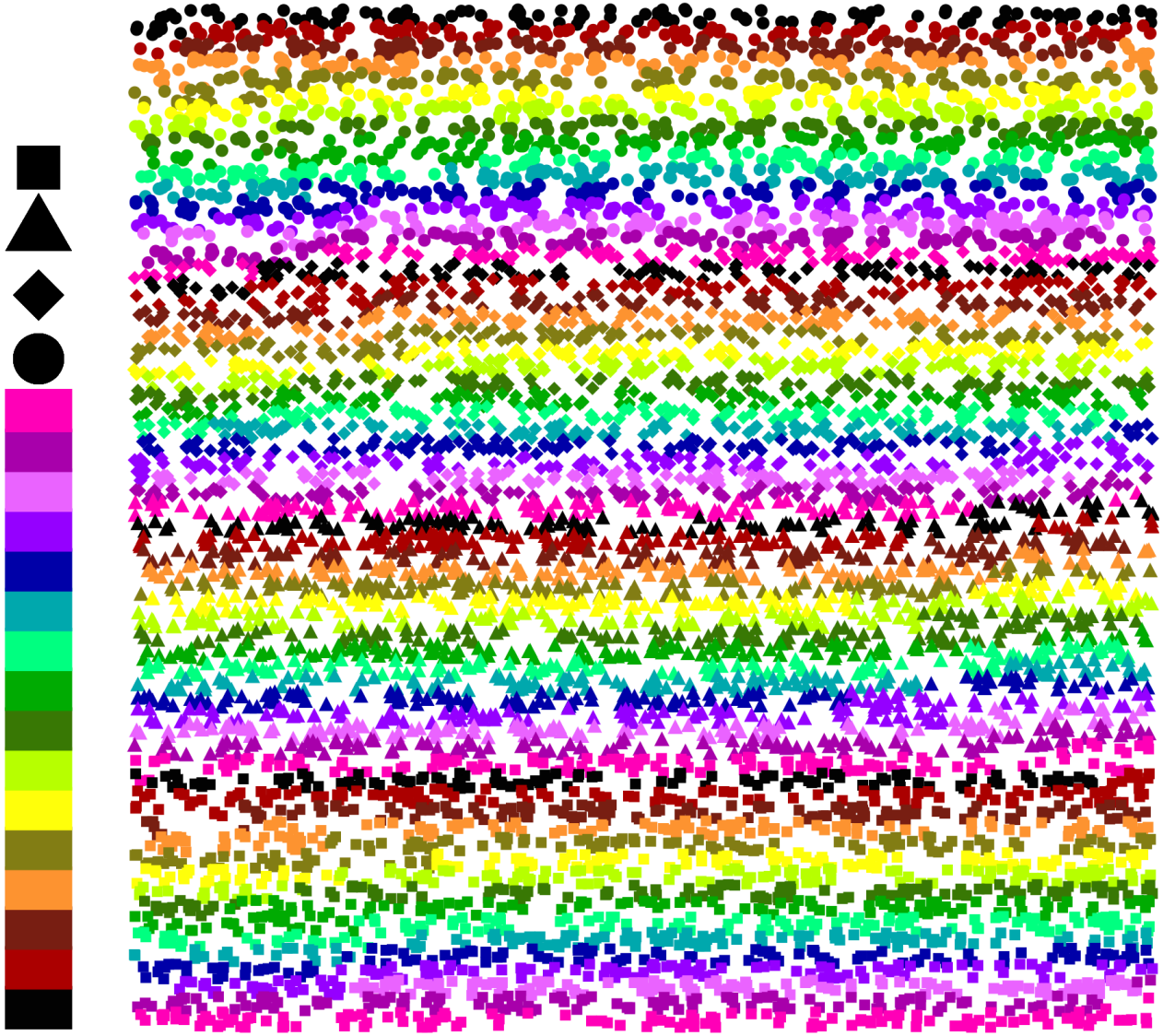
Space-Filling Curves for E and ρ: Cache Misses Results



Number of Level 3 cache misses[10], for 50,000,000 particles, with a 128×128 mesh, $\Delta t = 0.1$. Architecture: Intel Haswell (2013).

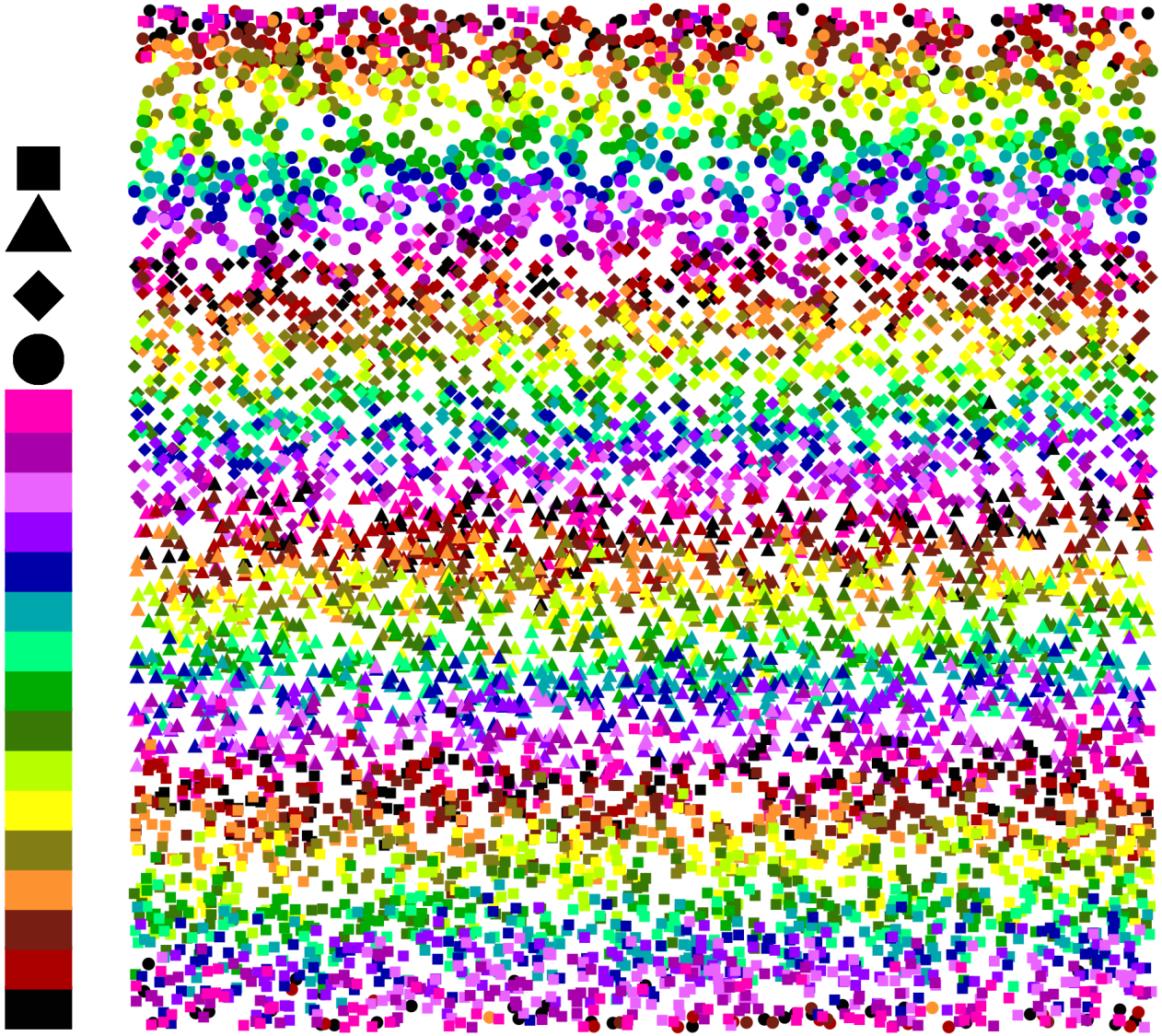
[10] Perf. Application Prgm. Interface (PAPI): http://icl.cs.utk.edu/papi

Space-Filling Curves: The 4-Figures Proof (64 × 64 mesh)



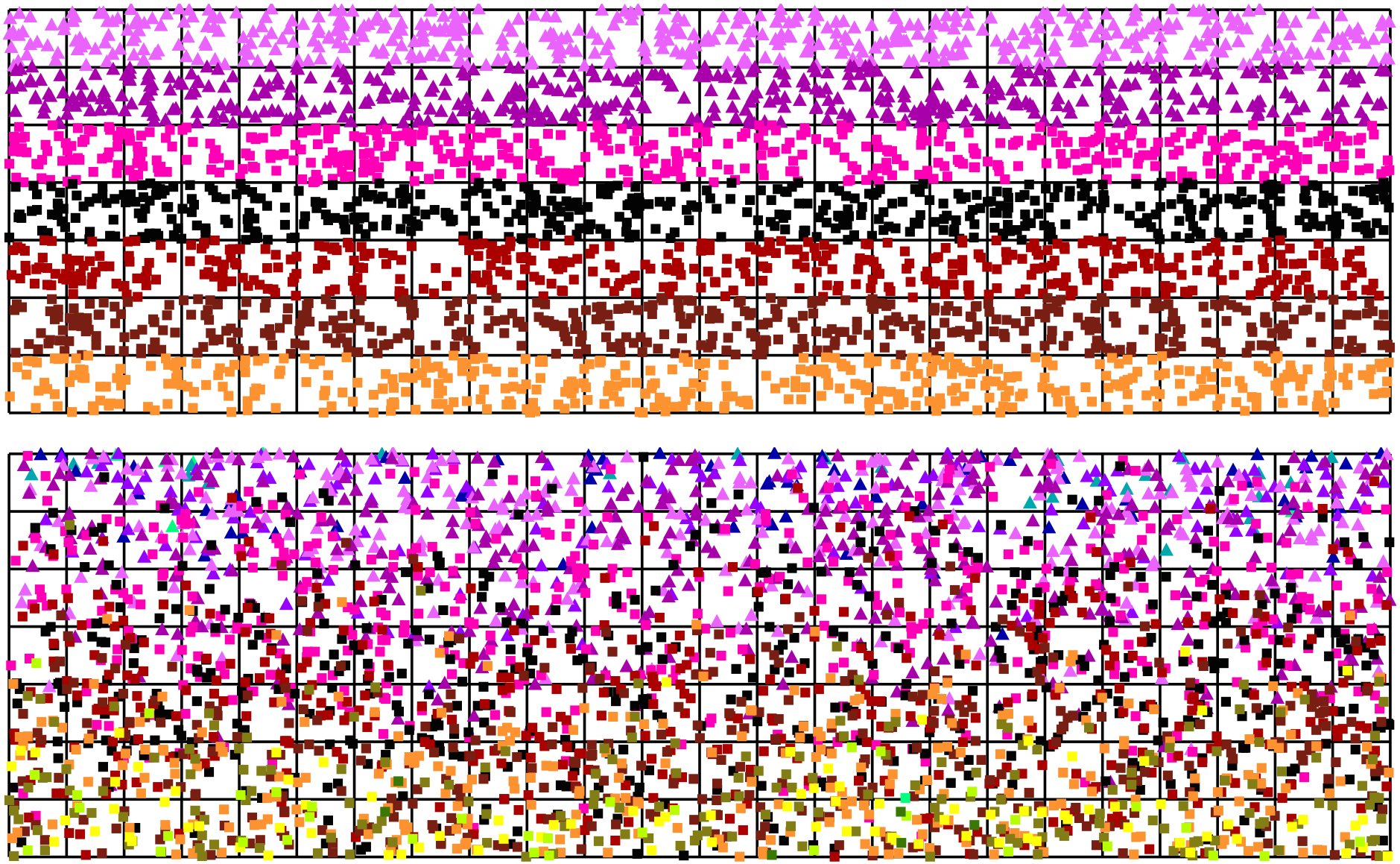
Row-Major: iteration 0

Space-Filling Curves: The 4-Figures Proof (64×64 mesh)



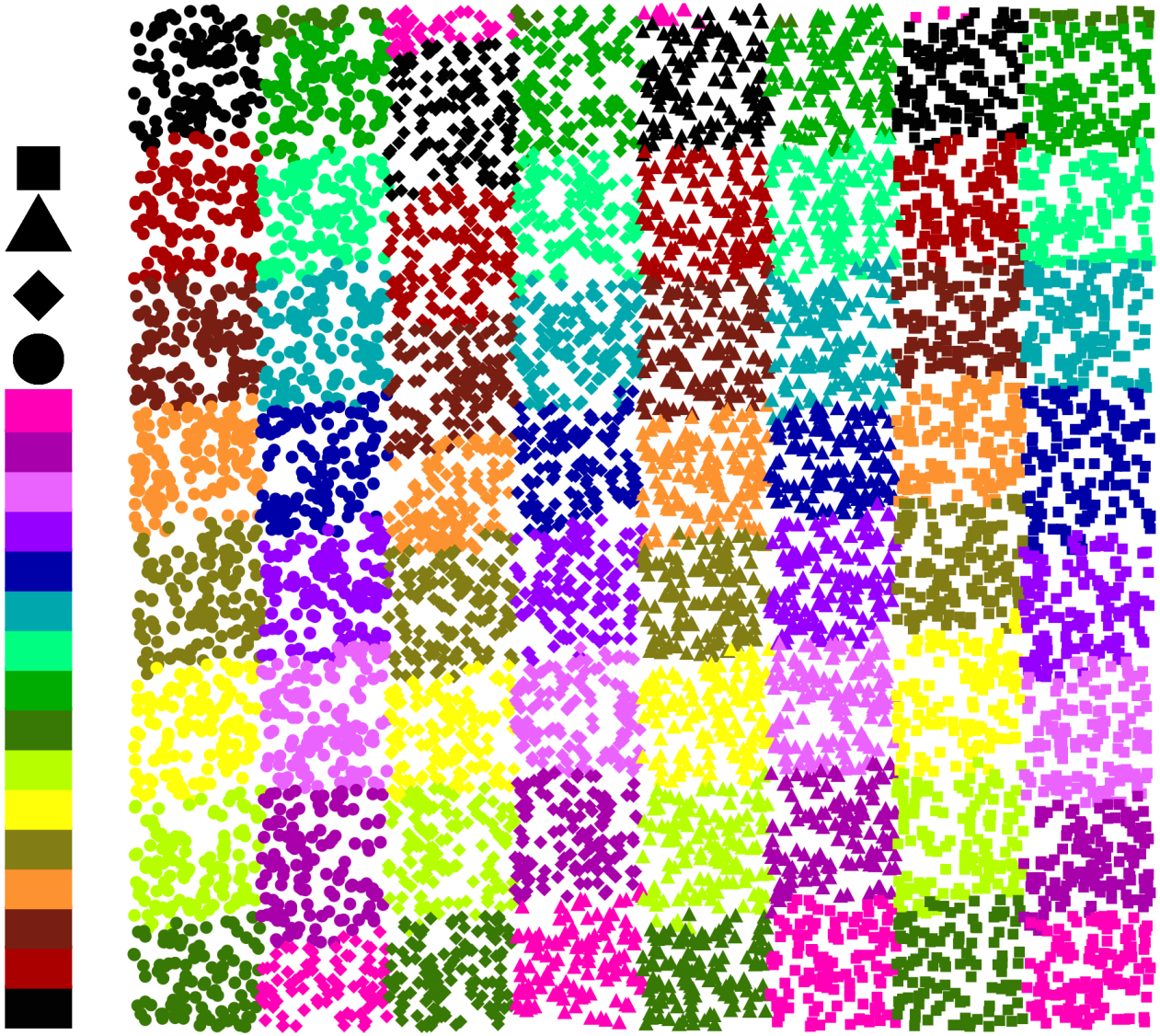
Row-Major: iteration 3

Space-Filling Curves: The 4-Figures Proof (64 × 64 mesh)



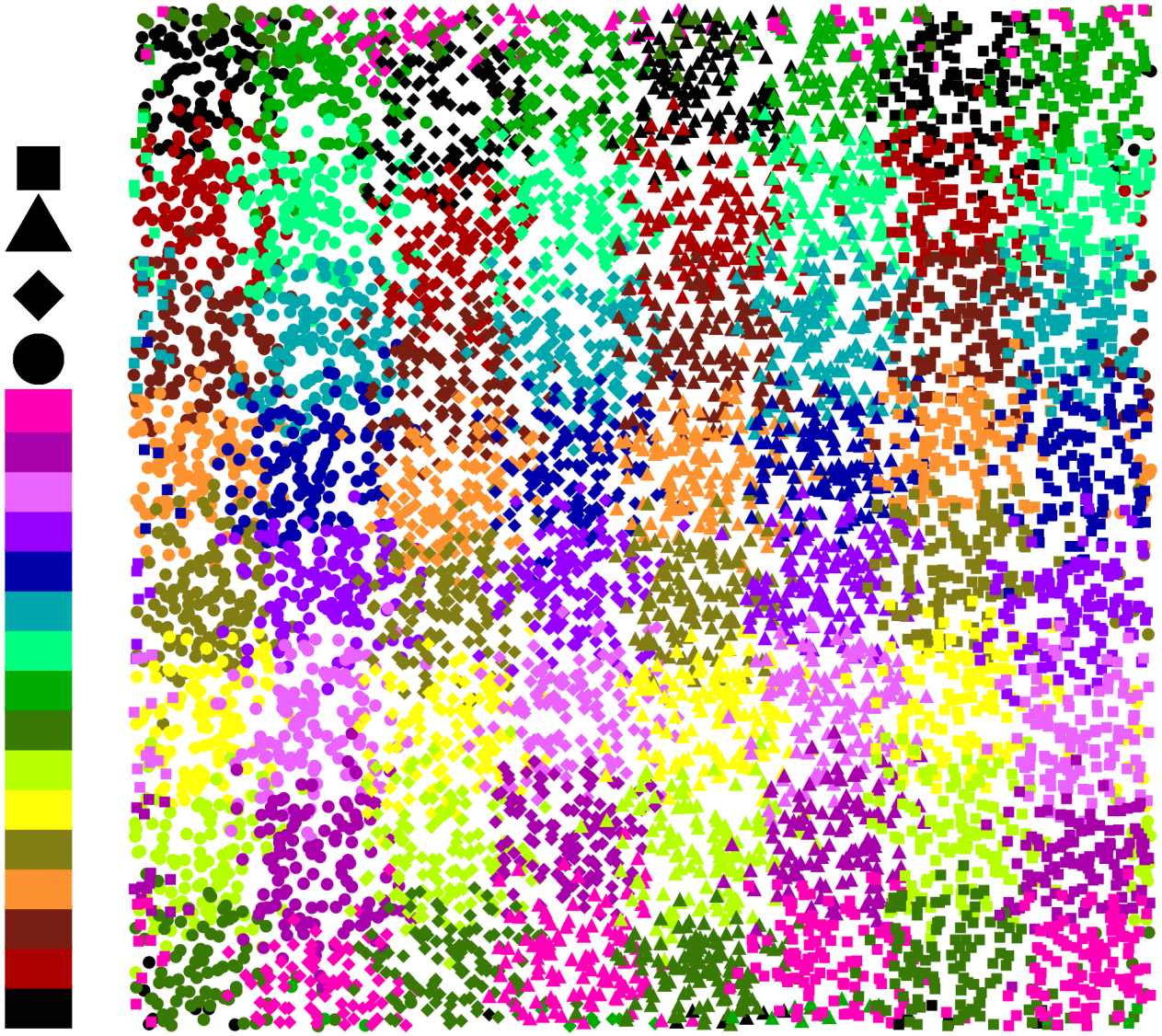
Row-Major: comparison iteration 0 / iteration 3 (zoom)

Space-Filling Curves: The 4-Figures Proof (64 × 64 mesh)



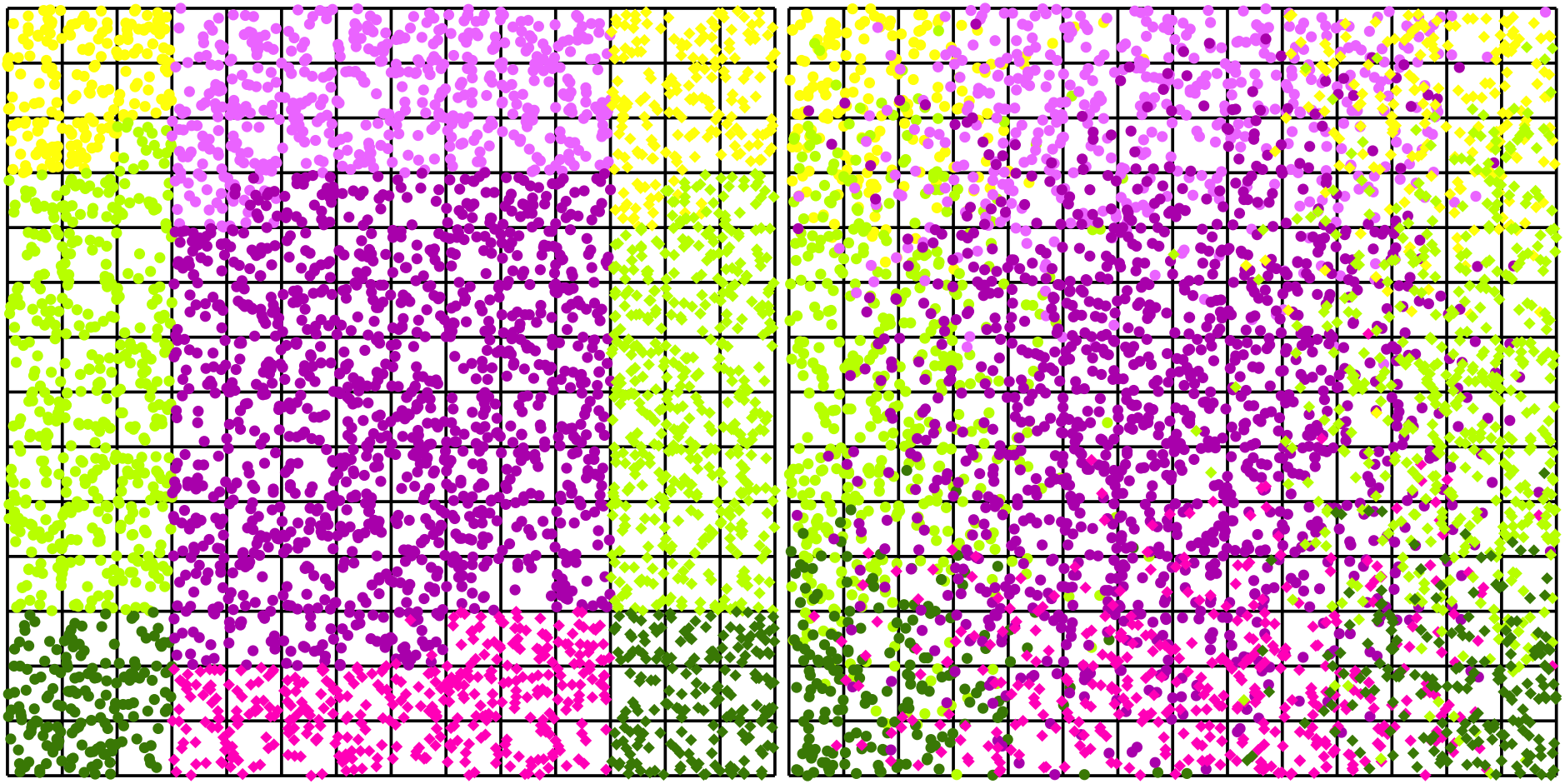
L4D: iteration 0

Space-Filling Curves: The 4-Figures Proof (64×64 mesh)



L4D: iteration 3

Space-Filling Curves: The 4-Figures Proof (64×64 mesh)



L4d: comparison iteration 0 / iteration 3 (zoom)

Hybrid Parallel and Vectorized Code

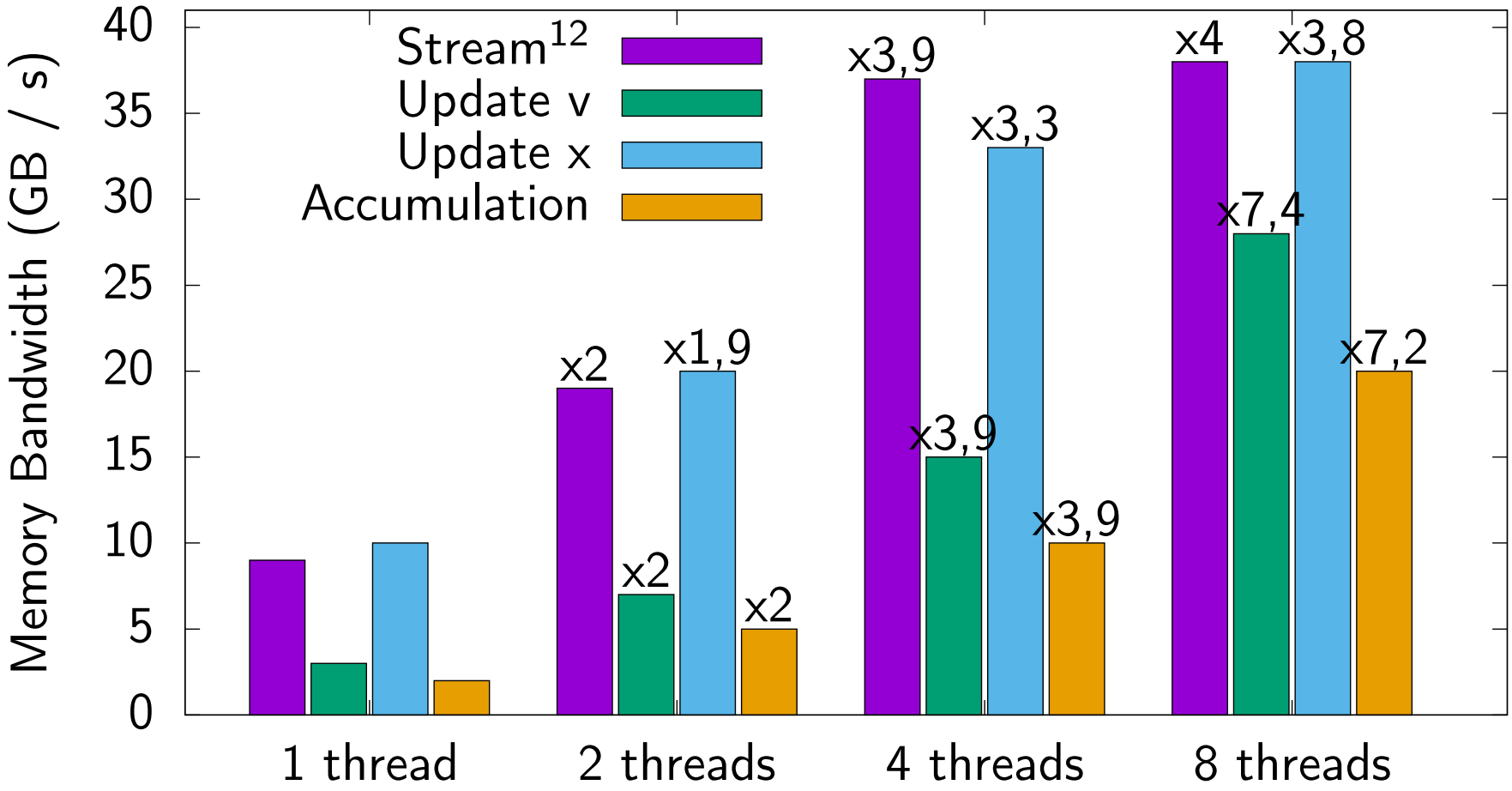
- Among MPI processes: particle decomposition (each process keeps a constant subset of the particles and the whole grid)
- Advantage: computations are automatically balanced
- Drawback: communication overhead (`MPI_ALLREDUCE`) when using more than a few thousand cores

- On one socket: OpenMP pragmas
  `#pragma omp for` for update-velocities and update-positions
  `reduction(+:rho[0:ncx*ncy][0:4])`[11] for accumulation
  out-of-place counting sort

- On one core: code efficiently and automatically vectorized by `icc` and `gcc` (no function calls, `if`s replaced by bitwise ands when the grid size is a power of 2), that led to 75 million particles processed/second on Intel Haswell (2013).

[11] Since OpenMP 4.5 in C.

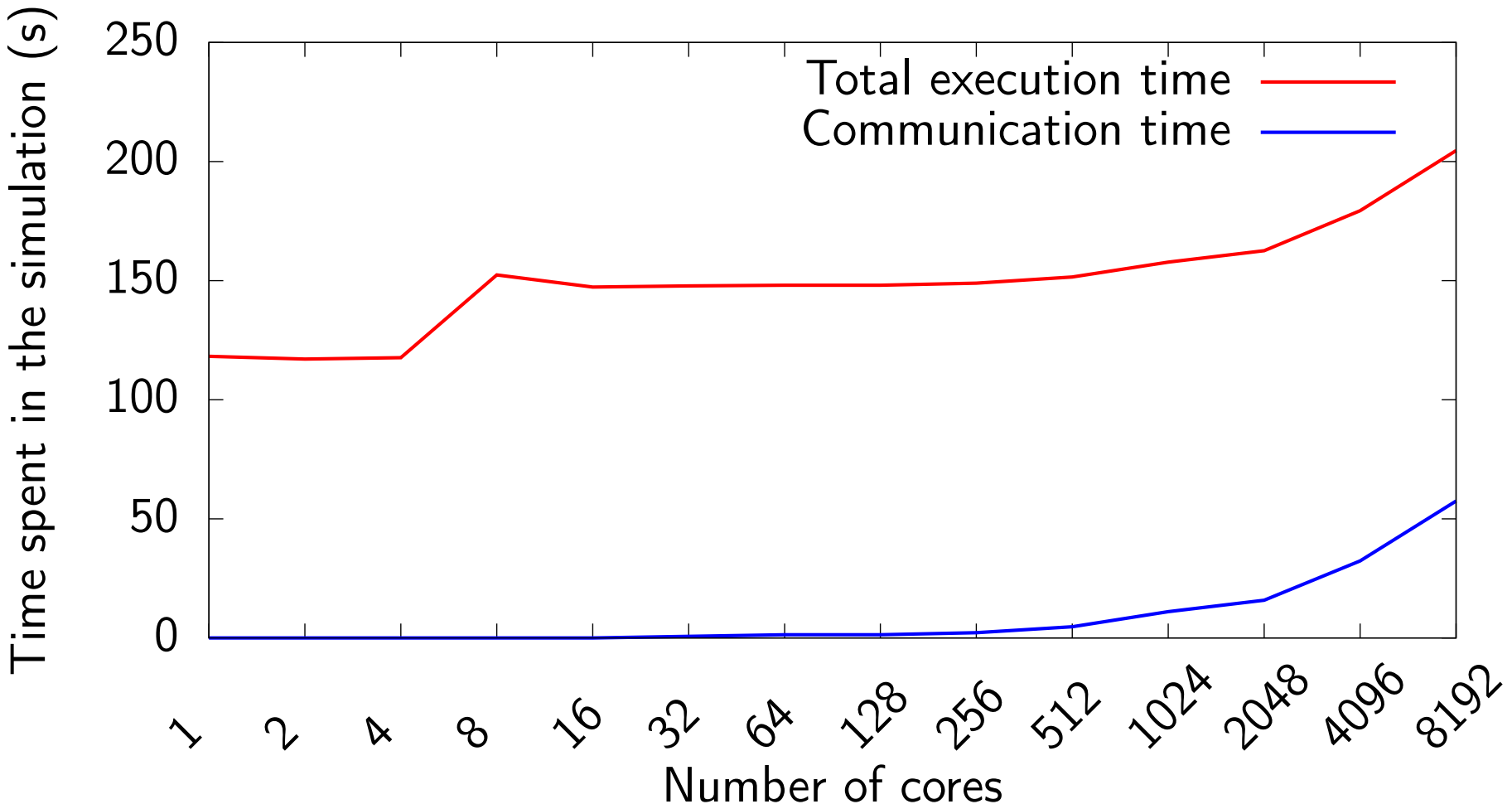
Strong Scaling on one Socket (8 Cores, 4 Memory Channels)



Strong scaling for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations): 95% of reachable scalability. Architecture: Intel Sandy Bridge EP (2011).

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

Weak Scaling on 8,192 Cores

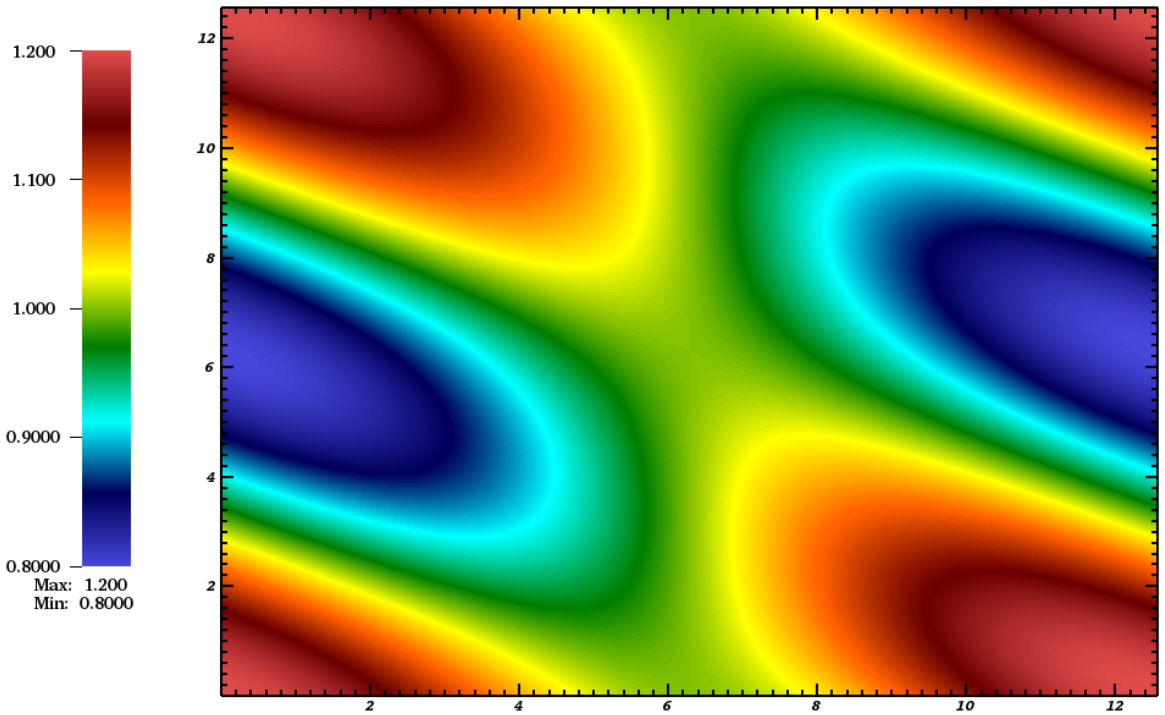


Weak scaling for a 128 x 128 grid, 50 million particles per core, 100 iterations simulation (sorting every 50 iterations): 75% parallel efficiency. Architecture: Intel Sandy Bridge EP (2011).

Two-stream instability

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.05$, initial condition[13]:

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



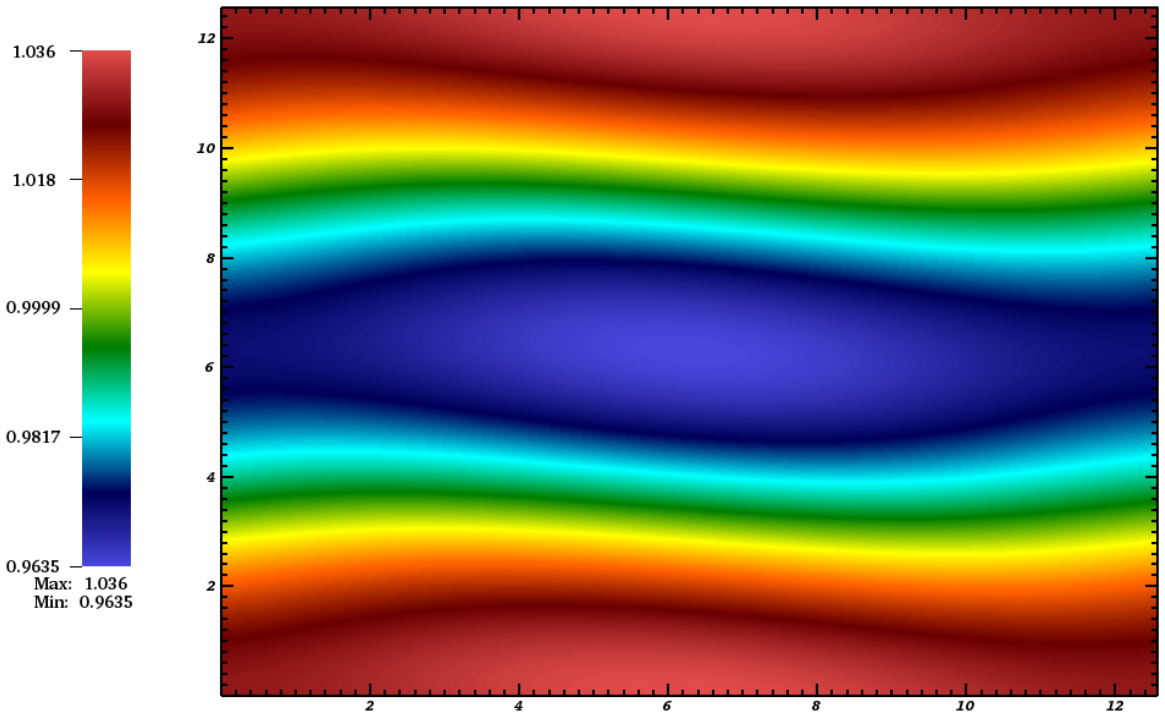
Iteration 0

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

Two-stream instability

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.05$, initial condition[13]:

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



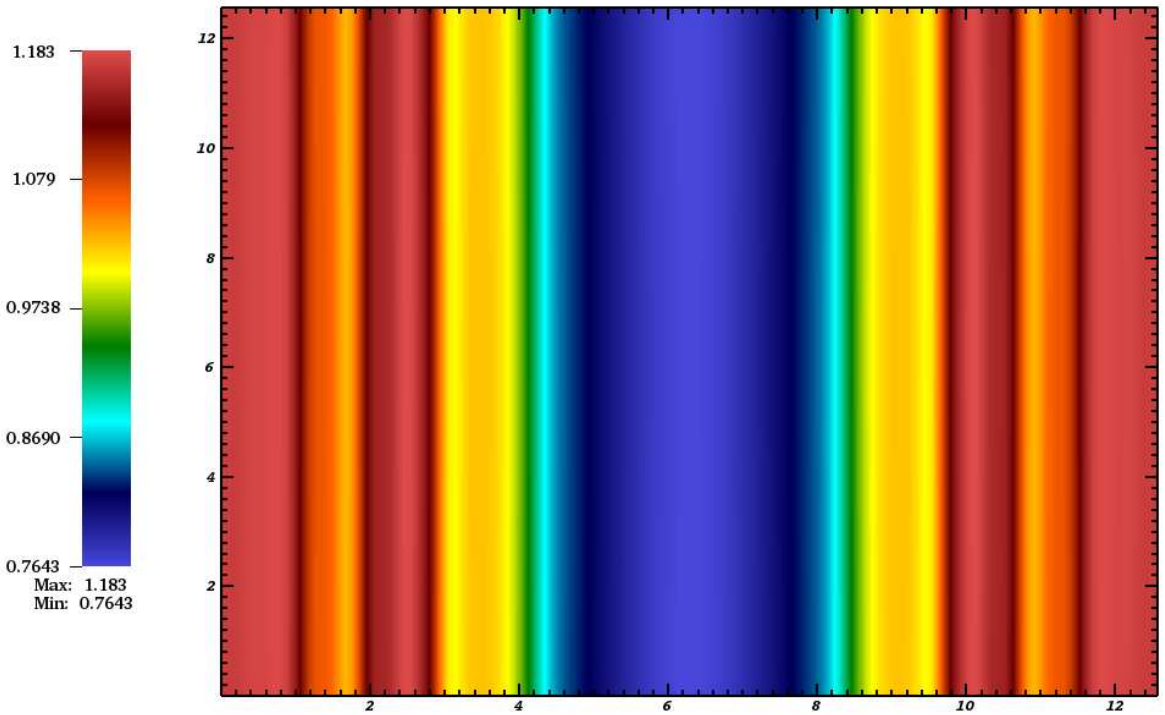
Iteration 100

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

Two-stream instability

$(x, y) \in [0; 4\pi)^2$, 128×128 grid, $\Delta t = 0.05$, initial condition[13]:

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



Iteration 1,000

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

Conclusions - Outlook

- efficient PIC code with standard numerical schemes
  - -36% in cache misses (space-filling curves)
  - -42.8% in total time when considering all optimizations
  - 75% parallel efficiency even with simple parallelization
- future outlook
  - instead of sorting the particle arrays, change the data structure to keep them always sorted (other optimizations possible[14,15])
- future future outlooks
  - port the code to Many Integrated Core (MIC) architecture like Intel Knights Landing (KNL)
  - port the code to 3d with domain decomposition (with load balancing)
  - use realistic physical parameters (geometry, equations)
  - use higher-order numerical methods (allows the use of less grid cells and/or less time steps)

[14] Nakashima, Summura, Kikura & Miyake (IPDPS 2017 - Session 6)

[15] Barsamian, Charguéraud & Ketterlin (2017)

ybarsamian@unistra.fr