Efficient Data Layouts for a Three-Dimensional Electrostatic Particle-in-Cell Code

Yann Barsamian[a,b,*], Sever A. Hirstoaga[b,c], Éric Violard[a,b]

[a] *Université de Strasbourg, CNRS, ICube UMR 7357, F-67412 Illkirch, France*
[b] *Inria Nancy - Grand Est, F-54600 Villers-lès-Nancy, France*
[c] *Université de Strasbourg, CNRS, IRMA UMR 7501, F-67084 Strasbourg, France*

Abstract

The Particle-in-Cell (PIC) method is a widely used tool in plasma physics. To accurately solve realistic problems, the method requires to use trillions of particles and therefore, there is a strong demand for high performance code on modern architectures. The present work describes performance results of `Pic-Vert`, a hybrid OpenMP/MPI and vectorized three-dimensional electrostatic PIC code.

The code simulates 3d3v Vlasov-Poisson systems on Cartesian grids with periodic boundary conditions. Overall, it processes 590 million particles/second on a 24-core Intel Skylake architecture, without hyper-threading (25 million particles per second per core).

The paper presents extensions in 3d of our preliminary 2d results [1], with highlights on the difficulties and solutions proposed for these extensions. Specifically, our main contributions consist in proposing a new space-filling curve in 3d (called L6D) to improve the cache reuse and an adapted loop transformation (strip-mining) to achieve efficient vectorization. The analysis of these optimization strategies is performed in two-stages, first on a 24-core socket and second on a super-computer, from 1 to 3,072 cores, demonstrating significant performance gains and very satisfactory weak scaling results of the code.

*Corresponding author
Email addresses: `ybarsamian@unistra.fr` (Yann Barsamian), `sever.hirstoaga@inria.fr` (Sever A. Hirstoaga), `violard@unistra.fr` (Éric Violard)

May 23, 2018

Keywords: data structures, space-filling curves, SIMD architecture, hybrid parallelism, strong and weak scaling, three-dimensional Particle-in-Cell simulation, plasma physics

1. Introduction

Implementing a Particle-in-Cell (PIC) method is an important step of many applications in the field of computational science. Although particle methods are usually characterized by low accuracy, they are able, when using a very large number of particles, to adequately reproduce complicated phenomena, for instance in plasma physics [2, 3]. However, in such a situation, a naive implementation can quickly exhibit memory bottlenecks since the overall cost is dominated by data motion and not by computation [4].

Therefore, a lot of research efforts were recently devoted towards more adapted PIC implementations that utilize efficiently modern super-computing resources [4, 5, 6, 7, 8, 9]. Despite these significant advances, targeting realistic applications in plasma physics is still challenging, due to the complex multi-scale six-dimensional problems to be solved in the phase space. In this direction, our objective is to develop a massively parallel and highly scalable PIC code, in view of physically meaningful realistic simulations.

First steps were achieved in [10, 1] where we built a hybrid parallel and vectorized PIC code for a Vlasov-Poisson model in a two-dimensional (2d) physical space. Several data structures for particles and for grid quantities were analyzed in order to enhance data locality and to reduce the execution time with respect to a classic code. More precisely, we showed in [1] that a structure of arrays and a L4D space-filling curve lead to an efficient ordering of the particles and of the electric field and the charge density, respectively. In addition, the performance of parallelization of the loops through distributed and shared memory paradigms was assessed in tandem with the memory channels.

In the present contribution we extend to three-dimensions (3d) for the physical space the PIC code in [1] for simulating electrostatic plasma. Even though

reduced 2d models are used in the literature to gain insights about the main behavior of the plasma, full 3d simulations clearly improve the realism of the physical description. Moreover, in some situations, reducing the dimensionality is not even possible and thus a 3d simulation is unavoidable. However, with the aim of keeping a satisfactory accuracy of the computed solutions in this case, we need to consider the significant increase in the amount of data to be processed.

A first obvious difficulty of the extension from 2d to 3d simulations is the need for more storage for the grid quantities [2, 3]. If, in a 2d simulation, keeping the whole grid quantities in the cache memory is still possible, this target is much more difficult or even impossible to achieve for a fine grid in the 3d case. Secondly, a 3d simulation requires more data traffic and computations. Specifically, passing from 4 grid points to 8 points for the interpolation and accumulation steps (when using a linear approach) becomes more difficult to handle, both for the data flow and for the vectorization. In addition, the difficulty increases significantly if higher order approximations within these steps are in use. The same challenges need to be addressed when vectorization is used for the updating positions step.

The paper is organized as follows: in Section 2 we present the basic kinetic model for the plasma, we detail the steps of the PIC implementation, introduce the related work and explain our contributions. In Section 3 we detail the code optimizations and we present the performance results on 24 cores with OpenMP only. In Section 4 we show the scalability of the code on up to 3,072 cores of the supercomputer Marconi. Section 5 summarizes the work and presents some future directions.

2. PIC overview for the 3d Vlasov-Poisson model

2.1. Description of the problem

A PIC method simulates a plasma by integrating self-consistently the trajectories of charged particles with fields that are generated by the particles themselves [2, 3]. In the case where there is no other external field and the

Parameters

N: number of particles.

$ncx \times ncy \times ncz$: number of grid cells.

Δt: time step.

f_0: initial distribution function.

q and m: particle charge and mass.

Variables

$particles[N]$: set of particles, with position $\mathbf{x_p}$ and velocity $\mathbf{v_p}$.

$\rho[ncx][ncy][ncz]$: charge density.

$\mathbf{E}[ncx][ncy][ncz]$: electric field.

Algorithm

1 Randomly initialize N particles following f_0
2 Compute initial ρ and $\mathbf{E}$
3 **Foreach** time step — Leap-frog
4 **If** (*condition*), **then**
5 Sort the particles
6 Set all cells of ρ to 0
7 **Foreach** particle
8 Interpolate $\mathbf{E}$ to $\mathbf{x_p}$ — Stored in $\mathbf{E_p}$
9 Update $\mathbf{v_p}$ — $\mathbf{v_p}$ += $\frac{q}{m}\Delta t\,\mathbf{E_p}$
10 Update $\mathbf{x_p}$ — $\mathbf{x_p}$ += $\Delta t\,\mathbf{v_p}$
11 Accumulate charge from $\mathbf{x_p}$ on ρ
12 Compute $\mathbf{E}$ from ρ — Poisson solver

Figure 1: High-level description of the Particle-in-Cell (PIC) method.

self-consistent magnetic field is neglected, this relies on solving the following Vlasov-Poisson system:

$$\begin{cases} \partial_t f + \mathbf{v}\cdot\nabla_{\mathbf{x}} f + \dfrac{q}{m}\mathbf{E}\cdot\nabla_{\mathbf{v}} f = 0 & \text{Vlasov} \\ f(\mathbf{x},\mathbf{v},0) = f_0 & \\ -\Delta\phi = \dfrac{\rho}{\epsilon_0} & \text{Poisson} \end{cases}$$

where

$$\rho(\mathbf{x},t) = q\int f(\mathbf{x},\mathbf{v},t)dv \quad \text{and} \quad \mathbf{E}(\mathbf{x},t) = -\nabla\phi(\mathbf{x},t).$$

In the system above, $f = f(\mathbf{x},\mathbf{v},t)$ stands for the distribution of one species of particles (with charge q and mass m) in a six-dimensional phase space (three dimensions for positions and three dimensions for velocities), ρ stands for the charge density, and $\mathbf{E}$ for the self-consistent electric field. A PIC method consists in discretizing (sampling) the distribution function by a collection of macro-particles that move in the phase space following the characteristics of the Vlasov equation. Then, a PIC simulation follows four steps: accumulate on the spatial grid the particle charge, solve the Poisson equation to obtain the grid electric

field, interpolate this field to the particles, and finally push in time the particle positions and velocities.

In our PIC code, the particle positions and velocities are initialized randomly using the WELL generator [11]. The particles all have the same fixed weight. They are advanced in time with a leap-frog time stepping [2, Section 2.4] (second-order in time). The electric field is computed by solving the Poisson equation on a uniform Cartesian grid, by a Fourier method. Then, for the particle and force weighting we use the Cloud-in-Cell model [12] (first-order in space), meaning that eight neighboring grid points are used in the interpolation/accumulation steps for a particle in that cell. The PIC pseudo-code is detailed in Fig. 1.

Next we describe the two basic types of data on which the sequential implementation of the code is based.

2.2. Particle Data Structure

A particle is given by its position and velocity. While the velocity is classically represented by three real numbers, the position is identified in the present work with a single cell index i_{cell} and three normalized offsets within this cell. The advantages of this representation are exposed in [4, Section III-E]. In addition to the parameters explained in Fig. 1, we denote the physical space by $[x_{min}; x_{max}) \times [y_{min}; y_{max}) \times [z_{min}; z_{max})$, and the grid spacing by $\Delta x = (x_{max} - x_{min})/ncx$, $\Delta y = (y_{max} - y_{min})/ncy$ and $\Delta z = (z_{max} - z_{min})/ncz$. Thus, a particle positioned at $(x_{\text{physical}}, y_{\text{physical}}, z_{\text{physical}})$ is mapped on the grid at the position $(x, y, z) \in [0; ncx) \times [0; ncy) \times [0; ncz)$, where

$$x = \frac{x_{\text{physical}} - x_{min}}{\Delta x}, \quad y = \frac{y_{\text{physical}} - y_{min}}{\Delta y} \quad \text{and} \quad z = \frac{z_{\text{physical}} - z_{min}}{\Delta z}.$$

Then, we consider the integers

$$i_x = \texttt{floor}(x), \quad i_y = \texttt{floor}(y) \quad \text{and} \quad i_z = \texttt{floor}(z), \tag{1}$$

and the normalized offsets (which are real numbers in $[0; 1)$)

$$dx = x - i_x, \quad dy = y - i_y \quad \text{and} \quad dz = z - i_z. \tag{2}$$

The cell index i_{cell} is a number in $\{0, 1, \ldots, ncx \cdot ncy \cdot ncz - 1\}$, taken as the image of some one-to-one mapping depending on (i_x, i_y, i_z). For example, the row-major mapping

$$
\begin{aligned}
(i_x, i_y, i_z) \mapsto i_{\text{cell}} &= (i_x \cdot ncy + i_y) \cdot ncz + i_z \\
i_{\text{cell}} \mapsto &\begin{cases} i_x = \left\lfloor \frac{i_{\text{cell}}}{ncz \cdot ncy} \right\rfloor \\ i_y = \text{mod}\left(\left\lfloor \frac{i_{\text{cell}}}{ncz} \right\rfloor, ncy \right) \\ i_z = \text{mod}(i_{\text{cell}}, ncz) \end{cases}
\end{aligned} \tag{3}
$$

is commonly used in `C`. However, several bijection mappings will be analyzed in the following sections with the aim of improving the cache reuse.

Finally, a particle is stored in memory with 1 `int` (i_{cell}), 3 `floats` (dx, dy and dz) and 3 `doubles` (vx, vy and vz). In our code, these 7 attributes are stored in a Structure of Arrays (SoA), to enable efficient vectorization. In this framework, the update particle position step (line 10 in Fig. 1) can be accomplished in the following four sub-steps:

Update positions

1 Compute (i_x, i_y, i_z) from i_{cell}.

2 Update (x, y, z) using formula (2) and line 10 in Fig. 1.

3 Compute the new values of $(i_x, dx, i_y, dy, i_z, dz)$ using formulas (1) and (2).

4 Compute i_{cell} from (i_x, i_y, i_z).

The reason behind this approach stems from the fact that on modern architectures, it might be faster to make rapid computations than to store numbers. In this case, we clearly need fast algorithms for computing the bijection functions in sub-steps 1 and 4 above. For example, the bijection mapping in (3) can be computed very fast in both directions, which is not the case for all the mappings analyzed in this paper. In Section 3.2 we show for some bijection mappings that computing them is faster than storing the integers (i_x, i_y, i_z) in addition to i_{cell}.

2.3. E and ρ Data Structure

The standard 3d representation of the electric field E and of the charge density ρ stores their values at the grid points:

```
double Ex[ncx][ncy][ncz], Ey[ncx][ncy][ncz], Ez[ncx][ncy][ncz];
double rho[ncx][ncy][ncz];
```

In this case, the interpolation of the 3d arrays `Ex`, `Ey` and `Ez` requires access to memory locations that are not contiguous. A solution to partially overcome this problem consists in storing components of the field in only one array [5, Section IV, Case 3]:

```
double Exyz[ncx][ncy][ncz][3];
```

Unfortunately, this data structure still leads to non-contiguous accesses. This problem is solved by using a redundant one-dimensional array of coefficients [13, 10]:

```
double E_1d[ncx*ncy*ncz][24];
double rho_1d[ncx*ncy*ncz][8];
```

The redundant array `E_1d` stores for each cell the values of each of the three field arrays at the grid points on the eight corners of the cell, contiguously in memory. The array `rho_1d` similarly stores for each cell the values of the charge density on the corners of that cell. They take eight times more memory than the standard layout, but it has been shown that, for the charge density, it is more efficient because it opens the possibility to vectorize the accumulation step (line 11 in Fig. 1) [6, Section 4.1.2.]. We show in the sequel how to gain performance through cache hit improvements, both for E and ρ.

2.4. Related Work

In order to scan E and ρ as locally as possible, the particles are sorted by cell index periodically in time (the step in lines 4–5 in Fig. 1) [5, Section VI]. The cost of this step is made linear in N by using the counting sort [14, Section 8.2], since the number of cells is much smaller than the number of particles.

Space-filling curves can be used to enhance cache performances. Precise results were shown on applications with regular memory accesses such as linear

algebra [15], or with irregular accesses such as the n-body problem [16]. Nevertheless, none of these results can directly apply to a PIC code, which has irregular accesses over the arrays E and ρ and not the particle array itself, as in the n-body problem. In the previous contribution [1] we showed how space-filling curves can enhance the performance of a 2d PIC code. The present study is an extension to 3d of this work.

The space-filling curves are also of interest in particle codes at the inter-process level, to achieve load balancing when using domain decomposition [7] or to minimize communication between processes [17].

A rather popular technique on modern PIC codes is to organize the particles by so-called super-cells, *e.g.* PIConGPU [18], UPIC [8], ORB5 [9], PICADOR [19], PICSAR [6]. The L6D space-filling curve we propose can be seen as a technique leading to a similar organization of the particles: particles are kept together in memory also in a block-like fashion, cf. Fig. 6; nevertheless we do not apply a reordering at each time step, and when we reorder, we perform this operation by cell and not by super-cell.

2.5. Contributions

The main contributions of this paper are focused on the efficiency of a 3d PIC code on a multicore architecture. We also present results in the 2d setting that underline the difficulties when passing from 2d to 3d. Since it is well-known that PIC codes are memory-bound, in this article we propose two directions to reach memory efficiency:

- ⋆ Our previous work in 2d [1] showed that space-filling curves can optimize the cache performance. We design a new space-filling curve, the L6D, and we demonstrate that we achieved this aim in 3d by comparison with classic curves. In addition, we illustrate its efficiency by showing snapshots of real simulations in a two-dimensional setting.

- ⋆ We use a loop optimization, namely the strip-mining, in order to be able to vectorize the code and reduce memory transfers in 2d and 3d. By

studying the memory bandwidth of our code from 1 to 24 cores, we provide an argument for the necessity of the strip-mining strategy. Moreover, we report the different outcomes of this loop optimization when going from 2d to 3d and thus, how to adapt this approach to the 3d case in order to achieve performance.

3. Single Socket Optimizations

In this section we describe different strategies to gain performance in 3d simulations on a single socket of 24 cores. The present study stems from [1, Section IV] where we performed a similar analysis on a single core. Our new approach is motivated by the differences between the optimization strategies on single core and on shared-memory multiple cores, due to the different ratios between computational performance and memory bandwidth of the two configurations.

Note that in many cases, 3d simulations benefit from the same optimizations as 2d simulations [1]. However, we discuss in the sequel an additional optimization that behaves differently in 2d and in 3d.

All the simulations in this paper were conducted on the Marconi supercomputer, on which we were granted the use of 64 nodes with 2 sockets each. Each socket is an Intel Xeon Platinum 8160 @ 2.1 GHz (Skylake), with 96 GB of RAM, 6 memory channels, and 24 cores. Our `C` code was compiled using Intel C Compiler 17.0.4, using the FFTW3 library [20] for the Poisson solver. The update-positions and accumulate loops are vectorized thanks to OpenMP 4.0 `#pragma omp simd`.

The results shown in this section were all obtained with the test case presented in Table 1. In addition, we also simulated nonlinear Landau damping and two-stream instability test cases. Theoretical results which allow to verify the code are available in [2, 3]. Thus, we checked the numerical conservation of the total energy and the numerical evolution in time of the electric field.

Table 1: Test case for OpenMP optimizations.

Physical test case	Linear Landau damping [2, 5.15], initial distribution $\left(1+0.01\cos\left(\frac{x}{2}\right)\cos\left(\frac{y}{2}\right)\cos\left(\frac{z}{2}\right)\right)\frac{1}{(2\pi)^{3/2}}\exp\left(-\frac{\lvert\mathbf{v}\rvert^2}{2}\right)$
Spatial grid	$[0;4\pi)^3$ decomposed in 64^3 cells, periodic boundaries
Number of particles	1 billion
Number of iterations	100 (sorting every 10 iterations)
Time step	0.05
Particle crossing: averaged, per iteration	49.4% of the particles move 1 cell away, 0.00150% of the particles move 2 cells away
Architecture	24 cores on Intel Skylake

3.1. Loop fission [21, Section 9.3]

The first optimization we implemented is the loop fission. More precisely, the loop "**Foreach** particle" in Fig. 1 is broken into three parts: one loop to interpolate E and to update velocities, one to update positions, and one loop to accumulate the charge. There are two main reasons to use three loops instead of one: (a) we can efficiently vectorize the update-positions as a stand-alone loop and (b) a separate processing of the arrays of E and ρ in different loops leads to a better overall memory management.

This transformation leads to the pseudo-code shown in Fig. 2, and speeds up the code by 8.9%. Starting from now, we call "update-velocities loop" the loop which contains both the interpolation and the velocity update.

3.2. Data Structure and Data Layout for E and ρ

3.2.1. Introduction

The aim of this section is two-fold: (*i*) first, we compare the relative performance of the standard 3d data structure for the electric field E to the redundant one and (*ii*) second, we assess the impact of different ordering strategies on both the electric field E and the charge density ρ. In the sequel, we suppose that ρ is always stored using the redundant data structure that has been shown to

```
1  Foreach particle in particles,                    Update-velocities loop
2      Interpolate E to particle
3      Update the velocity
4  Foreach particle in particles,                    Update-positions loop
5      Update the position
6  Foreach particle in particles,                    Accumulate loop
7      Accumulate particle charge to ρ
```

Figure 2: Particle-in-Cell pseudo-code with loop fission.

be effective when using SIMD architectures since it enables vectorization of the accumulate loop [6, Section 4.1.2.].

In PIC codes memory accesses are a well-known major bottleneck. Each time that the code accesses a cell of E or ρ, a contiguous portion of the array is loaded into the cache: ideally, all the computations using these data cells should be performed while the information is still there, avoiding to reload them later from the main memory. In this respect, the contribution we describe in this section shows that using different memory layouts for the redundant data structure decreases the number of cache misses.

It should be noted that a periodic sorting of the particles needs to be applied to improve data locality, since particles are moving at each iteration. In this manner, two particles contiguous in memory are in the same grid cell and thus, they access the same E (resp. ρ) cell during the interpolation (resp. accumulation). Nevertheless, sorting at every iteration would be computationally expensive and therefore we have to find a memory layout of the cells such that the cache benefits from the sorting last as long as possible. More precisely, with the previous notations, we aim at constructing a mapping $(i_x, i_y, i_z) \mapsto i_{\text{cell}}$ such that, when a particle moves from a cell to another, the probability that its new cell-index i_{cell} is close to the old one should be high.

We remark that the mapping of the row-major ordering in equation (3) has advantageous data locality when a particle moves along the z-axis: if i_z increases

by one, the new cell-index also increases by one (except for particles on the top face of the grid), becoming exactly the index accessed by the following particles in the particle array. However, when a particle moves along the other axes, this favorable behavior is lost: if i_y (resp. i_x) increases by one, the cell-index changes by ncz (resp. $ncy \cdot ncz$) which implies cache misses for the values of E and ρ.

Four different strategies for ordering the cells have been tested. They are listed below from the least to the most computational-intensive, in terms of the computation of the mapping $(i_x, i_y, i_z) \mapsto i_{\text{cell}}$:

(i) Scan-order (or row-major order): the canonical `C` memory layout.

(ii) L6D-order, cf. Fig. 6, which is an extension of the L4D-order, cf. [15, Section 2.1.] and Fig. 5. In this work, we present new algorithms to efficiently convert from and to these orderings (both in 2d and in 3d).

(iii) Morton-order ($\mathcal{N}$-order) or Lebesgue-order (Z-order), cf. [22] and Fig. 3. Algorithms to convert from and to this ordering can be found in [23].

(iv) Hilbert-order, cf. [24] and Fig. 4. Algorithms to convert from and to this ordering can be found in [25].

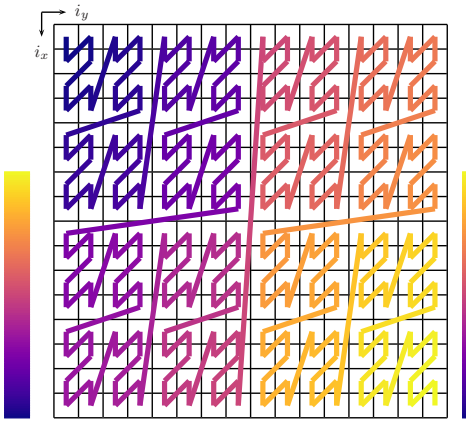


Figure 3: Morton layout of a 16 x 16 matrix.

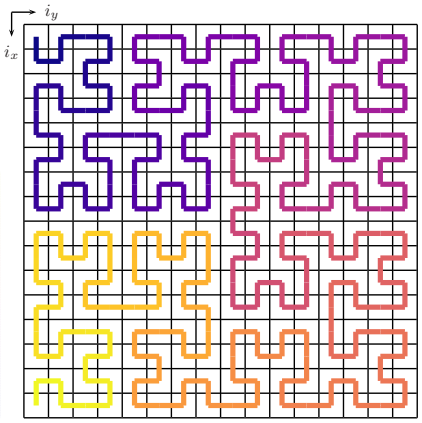


Figure 4: Hilbert layout of a 16 x 16 matrix.

```
1  for (i = 0; i < NB_PARTICLE; i++) {
2      x = i_x_from(i_cell[i]) + dx[i] + vx[i] * delta_t / delta_x;
3      y = i_y_from(i_cell[i]) + dy[i] + vy[i] * delta_t / delta_y;
4      z = i_y_from(i_cell[i]) + dz[i] + vz[i] * delta_t / delta_z;
5      i_x = (int)x - (x < 0.); // floor(x)
6      i_y = (int)y - (y < 0.); // floor(y)
7      i_z = (int)z - (z < 0.); // floor(z)
8      dx[i] = (float)(x - i_x);
9      dy[i] = (float)(y - i_y);
10     dz[i] = (float)(z - i_z);
11     i_cell[i] = i_cell_from((i_x + ncx) % ncx,
12                             (i_y + ncy) % ncy,
13                             (i_z + ncz) % ncz);
14 }
```

Listing 1: Sample `C` code for the update-positions loop. We have to add the grid sizes before applying the `%` operator because indices might become negative.

3.2.2. Bijection Algorithms

The aim of this section is to detail the formulas and their implementation needed for the application of space-filling curves in a PIC code. When using the cell index plus offset representation for the particles, the update-positions loop can be written as in Listing 1.

Explanations of the correctness of this loop, together with a way of removing the `* delta_t / delta_{x,y,z}` computations (lines 2–4) can be found in [1]. Below, we only focus on the `i_{x,y,z}_from` and `i_cell_from` functions.

In 2d, the mentioned functions are given in Listing 2 and are defined by

$$
\begin{aligned}
(i_x; i_y) \mapsto i_{\text{cell}} &= ncx \cdot \text{SIZE} \cdot \lfloor i_y/\text{SIZE} \rfloor + \text{SIZE} \cdot i_x + \text{mod}(i_y, \text{SIZE}) \\
i_{\text{cell}} \mapsto &\begin{cases} i_x = \lfloor \text{mod}(i_{\text{cell}}, ncx \cdot \text{SIZE})/\text{SIZE} \rfloor \\ i_y = \text{mod}(i_{\text{cell}}, \text{SIZE}) + \text{SIZE} \cdot \lfloor i_{\text{cell}}/(ncx \cdot \text{SIZE}) \rfloor \,. \end{cases}
\end{aligned} \tag{4}
$$

They can be set as macros or inline functions. In our code, the constant parameters are part of the macro arguments; they are here omitted to present a shorter code, as it will not affect performance. First, the tile size is given, as global parameter (line 1). Then, depending on the size of the grid, we have a constant which should be set at the beginning of the simulation (line 3): the

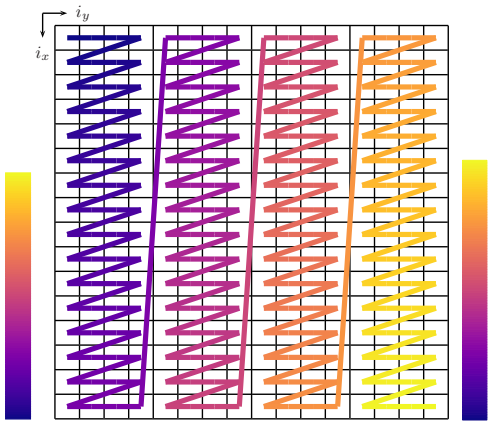


Figure 5: L4D layout of a 16 x 16 matrix, TILE_SIZE=4.

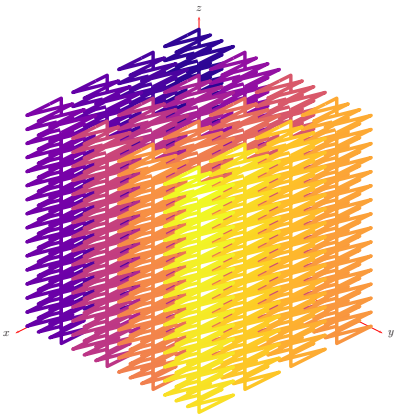


Figure 6: L6D layout of a 16 x 16 x 16 matrix, TILE_SIZE=4.

```
1  #define TILE_SIZE 8 // Depends on architecture.
2
3  int num_cells_per_column = ncx * TILE_SIZE;
4
5  #define i_cell_from(i_x, i_y) (TILE_SIZE * (i_x) + \
6      ((i_y) % TILE_SIZE) + num_cells_per_column * ((i_y) / TILE_SIZE))
7  #define i_x_from(i_cell) \
8      (((i_cell) % num_cells_per_column) / TILE_SIZE)
9  #define i_y_from(i_cell) (((i_cell) % TILE_SIZE) + \
10     TILE_SIZE * ((i_cell) / num_cells_per_column))
```

Listing 2: Efficient `C` code for L4D bijection functions.

number of cells per column gives the number of cells that are in a full column of width `TILE_SIZE`, 64 in Fig. 5.

The algorithm can be explained as follows: first, we have to know in which column the index is. This is computed as `i_y / TILE_SIZE`. The starting index of this column is thus `num_cells_per_column * (i_y / TILE_SIZE)`. Then, in this column of size `ncx * TILE_SIZE`, we recognize the row-major curve, which means we have to add `i_col * TILE_SIZE + j_col`. On the x-axis, the index is just `i_x`, and on the y-axis, the index is `mod(i_y, TILE_SIZE)`.

In 3d, the mentioned functions are given in equation (5) and Listing 3;

again, they can be set as macros or inline functions. In our code, the constant parameters are part of the macro arguments; they are here omitted. First, the tile size is given, as global parameter (line 1). Then, depending on the size of the grid, we have constants which should be set at the beginning of the simulation (line 3). The number of cells per tower gives the number of cells that are in a full tower of area `TILE_SIZE * TILE_SIZE`, 256 in Fig. 6. The number of cells per wall gives the number of cells that are in a full row of towers. In Fig. 6, there are 4 towers per wall (there are 4 towers of size `TILE_SIZE` $= 4$ in a row of $ncx = 16$ cells).

The explanation of the algorithm is similar to the 2d case. We have to know in which wall parallel to the (Oxz) plane the index is, then inside this wall in which tower it is, then this is a column-major ordering inside this tower.

$$
\begin{aligned}
(i_x; i_y; i_z) \mapsto i_{\text{cell}} = {} & ncz \cdot \text{SIZE}^2 \cdot \left(\lfloor i_x/\text{SIZE} \rfloor + \lfloor i_y/\text{SIZE} \rfloor \cdot \lceil ncx/\text{SIZE} \rceil \right) \\
& + i_z \cdot \text{SIZE}^2 + \text{mod}(i_y, \text{SIZE}) \cdot \text{SIZE} + \text{mod}(i_x, \text{SIZE})
\end{aligned}
$$

$$
i_{\text{cell}} \mapsto \left\{ \begin{array}{l}
i_x = \text{mod}(i_{\text{cell}}, \text{SIZE}) + \text{SIZE} \\
\qquad \cdot \left\lfloor \text{mod}(i_{\text{cell}}, ncz \cdot \text{SIZE}^2 \cdot \lceil ncx/\text{SIZE} \rceil)/(ncz \cdot \text{SIZE}^2) \right\rfloor \\
i_y = \text{SIZE} \cdot \left\lfloor i_{\text{cell}}/\left(ncz \cdot \text{SIZE}^2 \cdot \lceil ncx/\text{SIZE} \rceil\right) \right\rfloor \\
\qquad + \left\lfloor \text{mod}(i_{\text{cell}}, \text{SIZE}^2)/\text{SIZE} \right\rfloor \\
i_z = \left\lfloor \text{mod}(i_{\text{cell}}, ncz \cdot \text{SIZE}^2)/(\text{SIZE}^2) \right\rfloor
\end{array} \right. \tag{5}
$$

3.2.3. Results

We present in Tables 2 and 3 the performance gains when using the space-filling curves described in the previous section. Our preceding contribution [1] only showed results on one core. On modern architectures, there are usually more cores than memory channels: it is thus not straightforward to extrapolate the one core results on the full multicore architecture; we therefore show here results on the full processor. Moreover, additional arrays to store the indices i_x and i_y were used in [1]. We show now that additional gains can be obtained with efficient computations of the bijection functions. In the tables, "arrays" means that we use additional arrays to store the indices, otherwise we recompute them.

```
#define TILE_SIZE 8 // Depends on architecture.
#define SQR_TILE_SIZE (TILE_SIZE * TILE_SIZE)

int num_cells_per_tower = ncz * SQR_TILE_SIZE;
int num_cells_per_wall  = num_cells_per_tower * \
    ((ncx + TILE_SIZE - 1) / TILE_SIZE); // ceiling(ncx/TILE_SIZE)

#define i_cell_from(i_x, i_y, i_z)                                    \
    (((i_x) / TILE_SIZE) * num_cells_per_tower +                      \
    ((i_y) / TILE_SIZE) * num_cells_per_wall + (i_z) * SQR_TILE_SIZE + \
    ((i_y) % TILE_SIZE) * TILE_SIZE + ((i_x) % TILE_SIZE))
#define i_x_from(i_cell) (((i_cell) % TILE_SIZE) + \
    (((i_cell) % num_cells_per_wall) / num_cells_per_tower) * TILE_SIZE)
#define i_y_from(i_cell) (((i_cell) / num_cells_per_wall) * TILE_SIZE + \
    ((i_cell) % SQR_TILE_SIZE) / TILE_SIZE)
#define i_z_from(i_cell) (((i_cell) % num_cells_per_tower) / \
    SQR_TILE_SIZE)
```

Listing 3: Efficient `C` code for L6D bijection functions.

Table 2: 2d Space-filling Curves Timings

	Up. v	Up. x	Acc.	Sort	Total
2d standard	59.0	39.8	41.9	28.6	171.1
Row-major	63.6	39.7	42.8	28.6	176.8
L4D arrays	57.6	48.2	33.5	41.1	182.7
Morton arrays	60.2	48.0	29.4	40.7	180.7
Hilbert arrays	64.9	49.6	30.7	40.5	193.1
L4D	57.5	40.0	32.0	28.6	160.5
Morton	59.3	39.8	29.8	28.4	159.7
Hilbert	59.0	323.7	33.6	28.6	452.3

Time spent in the different loops (in seconds). Test case: Landau damping 2d on a $[0; 4\pi)^2$ grid decomposed in 512^2 cells, 1 billion particles, 100 iterations (sorting every 20 iterations), $\Delta t = 0.1$, 24 Skylake cores.

We focus on three meaningful comparisons in Tables 2 and 3. The first one is on the data structure: is it beneficial to use the redundant one for the E

Table 3: 3d Space-filling Curves Timings

	Up. v	Up. x	Acc.	Sort	Total
3d standard	126.7	55.3	31.5	21.5	235.8
Row-major	92.6	55.3	31.5	21.4	201.7
L6D arrays	92.8	79.0	30.4	29.5	232.6
Morton arrays	96.5	79.0	30.3	27.5	234.2
Hilbert arrays	95.3	80.4	31.1	26.9	234.8
L6D	85.5	55.5	29.9	20.9	192.9
Morton	89.4	56.7	33.5	19.8	200.3
Hilbert	87.3	244.4	29.2	20.3	382.2

Time spent in the different loops (in seconds). Test case in Table 1.

arrays? The only point where the code changes is in the update-velocities loop. We see in the tables that in 2d, it is detrimental to use it if we stick to the row-major curve, but it is already beneficial with the canonical curve in 3d. We recall that we use here many particles per grid cell, and that when using only a few particles per cell, the redundant data structure is not a good choice, see Section 3.4 for a detailed comparison.

The second comparison concerns the data layout. Is it possible to obtain notable gains by changing the ordering of the grid cells? There are two places in the code where the changes might become significant: in the interpolation and in the accumulation. This time, we can answer positively. Yes, by taking another order than the canonical one, we can save time (thank to a reduction in the cache misses [1, Table 2]). In 2d, the L4D and Morton curves seem to give similar and optimal timings, while in 3d, the L6D curve allows additional gains and seems to be the best one.

The last comparison is on the particle data structure needed for the non-canonical orderings. We can either store the indices in additional arrays (here, arrays of `short int`), or re-compute them at each time step. When storing them, it requires more memory for the particles, therefore we need more time

in the update-positions loop and in the sorting step.

Last but not least, we see a surprising timing of the update-positions loop when using the Hilbert ordering without additional arrays. It is due to the fact that the computation (i_x, i_y, i_z) from i_{cell} is expensive and not vectorized. This is thus the only curve for which using additional arrays is profitable. Nevertheless, even with additional arrays, this curve does not improve performances compared to the standard layout, and consequently has to be discarded.

3.2.4. Additional Remarks

We have to choose carefully `TILE_SIZE` depending of the cache sizes. In our tests, `TILE_SIZE` $= 8$ led to the best timings. It can be replaced with other values, as long as they are not too large for the cache.

It should be noted that choosing a value of `TILE_SIZE` that does not divide the grid sizes is possible: then, there will be a few allocated cells that correspond to physical positions outside the boundaries and that will never be accessed.

As a side note, we can remark that if grid sizes are powers of two and if the architecture represents integers with two's complement, we can save some computations on each modulo operation (lines 11–13 in Listing 1). For example for the modulo in the x-axis, we can use the variable `int ncx_minus_one = ncx - 1` and then compute `mod(i_x, ncx)` as `i_x & ncx_minus_one`. This is more efficient than `(i_x + ncx) % ncx` on one core. However, when using the full 24-cores architecture, this small optimization brings no significant gains. The same goes for all the modulo computations in the L4D and L6D bijection functions, whenever we compute modulo a power of two.

We also note that it is possible to use space-filling curves without using the redundant data structure for E. We do not show here the corresponding results, but the conclusion is that when using a high number of particles per cell (as is our case) the redundant data structure turns out to be the best approach. This is clearly not true when using a low (*e.g.* less than a hundred) number of particles per cell.

In the previous paper [1], we showed that in 2d we could save up to 43%

cache misses on E and ρ. This was confirmed by measurements. We here give an intuition of why this works, in the particular case of our Landau damping test case, which is roughly isotropic. We show in Figs. 7, 8, 9 and 10 snapshots of the same 2d simulation, one using the row-major curve and the other using the L4D curve. For each ordering, two snapshots are taken: a first snapshot just after a sorting, and another one 3 iterations later. For each ordering, a coloring is applied on the particles to indicate which particles are close in memory. We will here focus on the black particles in the center of the row-major ordering just after the sorting, and on the deep purple particles in the center of the L4D ordering just after the sorting. Just after the sorting, the black particles occupy only the center cells. Three iterations later, they are spread on 7 as many cells (some move up to 3 cells up, some move up to 3 cells down - the ones that move horizontally do not incur the use of new cells). Just after the sorting, the deep purple particles occupy only the center cells. Three iterations later, they are spread on 3 as many cells (they are spread on a 14 x 14 square instead of a 8 x 8 square). This explains why, when fetching the E values and when updating the ρ values, the L4D curve will lead to less cache misses than the row-major one.

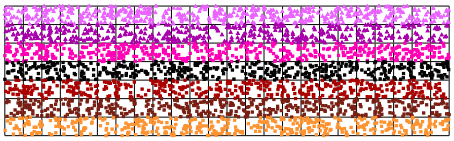


Figure 7: Row-major ordering, just after a sorting.

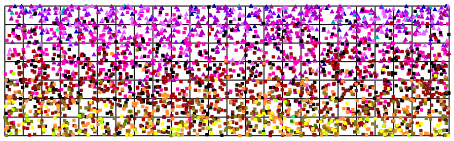


Figure 8: Row-major ordering, 3 iterations after a sorting.

In 3d, the explanations are similar, but harder to show on a figure.

3.3. Strip-mining [21, Section 9.8]

Even though the loop fission gives satisfactory results, the resulting code still needs to scan some particle arrays three times, thus putting a lot of pressure on the memory bus. A natural loop transformation that comes after is thus the strip-mining. Instead of having three loops each scanning all the particles, we split the particle arrays in sub-arrays of size k (where k has to be chosen,

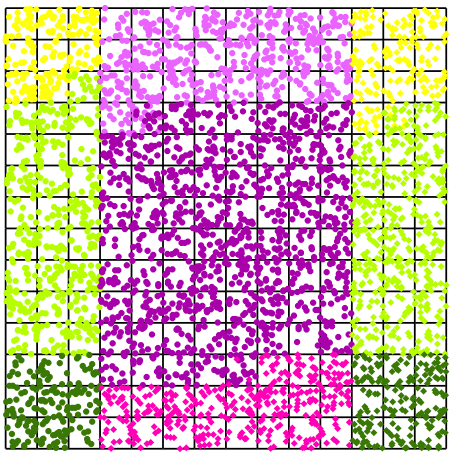


Figure 9: L4D ordering, just after a sorting.

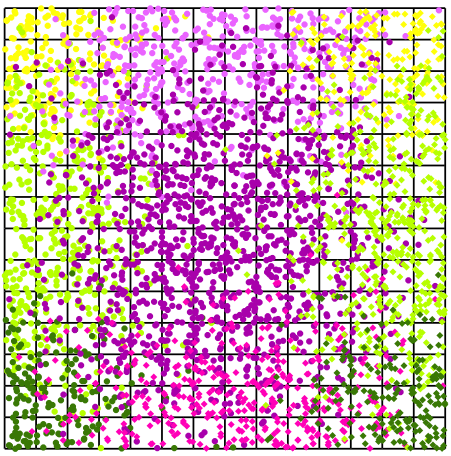


Figure 10: L4D ordering, 3 iterations after a sorting.

depending on the architecture), and have the three loops operate only on k particles. Thus, for each particle, instead of having to fetch its properties in the main memory for each loop, it is now possible to fetch its properties in the cache for the two last loops. This transformation leads to the pseudo-code shown in Fig. 11, and speeds up the code by 22% in 2d.

We can note that if we choose $k = 1$, we are back to the base code, which was not optimal. If we choose $k = N$, we are back to the previous code with loop fission. In our 2d experiments, choosing a strip-size k between 64 and 256 gives similar optimal results.

Unfortunately, in 3d this strip-mining does not improve performances. This is explained by the fact that the cache is filled with too many E values, thus the expected gain in performance coming from the cache reuse of the particle arrays is out of reach. Thus, in 3d, to be able to efficiently reuse the particle data, the strip-mining has to be done only on the two last loops. This transformation leads to the pseudo-code shown in Fig. 12, and speeds up the code by 12%.

In our 3d experiments, choosing a strip-size k between 32 and 128 gives similar optimal results.

This strip-mining technique was more or less already used in VPIC [4], which

```
1  Foreach subset of k particles in particles,
2      Foreach particle in this subset,
3          Interpolate E to particle
4          Update the velocity
5      Foreach particle in this subset,
6          Update the position
7      Foreach particle in this subset,
8          Accumulate particle charge to ρ
```

Figure 11: Particle-in-Cell pseudo-code with strip-mining.

```
1  Foreach particle in particles,
2      Interpolate E to particle
3      Update the velocity
4  Foreach subset of k particles in particles,
5      Foreach particle in this subset,
6          Update the position
7      Foreach particle in this subset,
8          Accumulate particle charge to ρ
```

Figure 12: Particle-in-Cell pseudo-code with strip-mining on the two last loops only.

advanced 4 particles at a time for vectorization ($k = 4$ with our notations). However, in VPIC, an additional assumption was made – none of those 4 particles should cross cell boundaries – and scalar code was generated for the particles that crossed cell boundaries. In our code, the update-positions loop is vectorized without exceptions.

3.4. Overall gains and comparisons

The optimizations presented in this section are summarized in Table 4. In this table, the baseline is a version of the code with the standard 3d data structure for E, the redundant one for ρ, and a loop hoisting [26, Section 2.3.5.3] already applied that, in particular, remove the `* delta_t / delta_{x,y,z}`

computations (lines 2–4 in Listing 1) [1]. The gains (in %) are computed with respect to the previous line of the table and the accumulated gains are computed with respect to the baseline.

- ⋆ Loop fission: better memory management for E and ρ, allows to vectorize the update-positions loop.
- ⋆ Redundant arrays for E together with appropriate use of space-filling curves: less cache misses in the interpolation and in the accumulation. We note that the redundant arrays are only useful when using at least a hundred particles per grid cell, see Table 5.
- ⋆ Strip-mining: allows to reuse particle data between loops.

Table 4: Gains of Different Optimizations

	Time (s)	Gains	Accum. gains
Baseline	258.7	0.0%	0.0%
+ Loop Fission	235.8	8.9%	8.9%
+ Space-filling curves (L6D)	192.9	18.2%	25.4%
+ Strip-mining	169.5	12.1%	34.5%

Total execution time, gains and accumulated gains. Test case in Table 1.

Overall, these optimizations result in 590 million particles processed per second, on 24 cores on Intel Skylake architecture, without hyper-threading (25 million particles per second per core), or 1.48 ns per particle per time step (35.47 ns per particle per time step per core).

Those performances are compared in Table 5 to another recent PIC code [9], solving the same equations with the same precision (both codes use double precision and first order interpolations). We ran simulation with the parameters chosen in this other paper, which differ from our previous ones. Simulations presented in that paper were conducted on the Piz Daint supercomputer consisting of 8 Sandy-Bridge cores @2.6 GHz with a theoretical memory bandwidth

Table 5: Comparison with Another Code

Code \ Nb. of particles	10^6	$16 \cdot 10^6$
Jocksch et al. work [9] (8 cores, 6.4 GB/s/core)	153.92	115.76
This paper (24 cores, 5.3 GB/s/core)	854.33	93.07

Time spent per particle per iteration per core, in nanoseconds. Test case [9]: 512 x 256 x 1 grid. Initial particle distribution uniform in space and velocity with $|v_{max}| = 1$.

of 51.2 GB/s (or 6.4 GB/s/core), and we recall that we used the Marconi supercomputer consisting of 24 Skylake cores @2.1 GHz with a theoretical memory bandwidth of 127.99 GB/s (or 5.3 GB/s/core). PIC codes being memory bound, comparing performances per core on those two architectures makes sense.

In those cases, there are only 7.6 particles per cell (when using 1 million particles), and 122 particles per cell (when using 16 million particles). The redundant data structure for E is a good choice only if there are a lot of particles per cell (in this paper, we use $3,815$ particles per cell for our test case in Table 1). Of course, when there are roughly as many particles as grid points, multiplying by 8 the data for E almost doubles the memory transfers. With such a low number of particles, using a redundant data structure for E is detrimental, and we would have to make another study for such a configuration. But with 122 particles per cell, the data layouts and code transformations presented in this paper become useful. At the scale of 1 billion particles, we recall that our code needs only 35.47 ns per particle per iteration per core.

4. Thread-level and process-level parallelism

4.1. Process-level parallelism

The state-of-the-art approach for parallelizing PIC simulations on distributed memory machines is to decompose the physical domain into smaller sub-domains and to assign the particles inside a sub-domain to a processor (among the wide literature, see *e.g.* [4, 7]). In grid-based simulations, this method was established to give good scaling, as long as the work due to the communications through the sub-domain boundaries remains small compared to the computations inside the sub-domains. However, the main drawback of this technique is the difficulty of maintaining the load balance.

In this work, we handle the process-level parallelism with particle decomposition instead of domain decomposition: during the whole simulation, every process holds a fixed amount of particles but it keeps track of the whole grid quantities. Thus, at every iteration, every process accumulates the charge density associated with its particles and an `MPI_ALLREDUCE` gives the total charge density. The Poisson equation is then solved by every process over the whole grid.

The main advantage of this method is its simplicity: the only communication is via `MPI_ALLREDUCE` for the reduction of the charge array and no particle has to move from one process to another during the simulation. Thus, all the computations are automatically work-balanced.

The bottleneck of this approach usually cited in the literature is that the scalability is highly limited by the global reduction step. Of course, when using a large number of processes, the communication becomes too costly. But in practice, if we use the full memory on each MPI process to put particles, this is not the main drawback. The main bottleneck of this approach is that in realistic simulations (using a grid bigger than our 64 x 64 x 64 grid), there are so many cells that the computations on one process will be inefficient, due to the high number of cache misses involved. This behavior is quite common for example in matrix computations, where you can see super-scaling behaviors on

large matrices, due to the large reduction of cache misses when computing only on sub-blocks. This is the reason why the technique we propose in this paper should be seen as only a small brick inside a more complex scheme: one should probably use domain decomposition on top of the efficient OpenMP algorithm we provided.

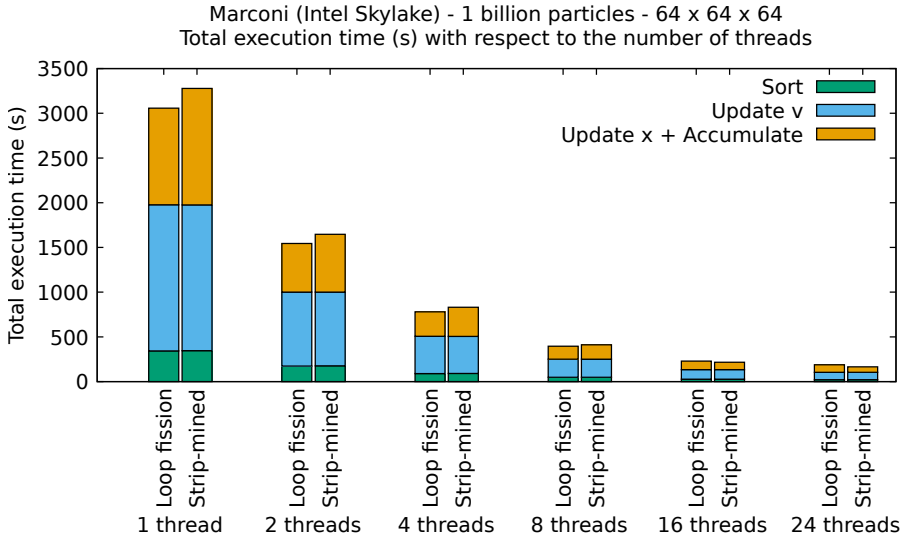
4.2. Strong scaling results on 24 threads with OpenMP

We next show a strong scaling up to 24 cores of two versions of our code. Fig. 14 shows a strong scaling of the code with loop fission (pseudo-code given in Fig. 2) and Fig. 15 show a strong scaling of our code with strip-mining (pseudo-code in Fig. 12).

Fig. 13 shows both codes on a single graph, for comparison. On 1 thread, the loop fission code is 6.7% faster, but on 24 threads the strip-mining code is 12% faster.

These figures illustrate the importance of having an efficient code on one core, but also the importance of precise performance analysis to enhance multicore efficiency. On the comparison graph, we see that up to 8 threads, the code with 3 loops performs better, then the code with strip-mining performs better. To understand why, we can look at the two other figures, where we show the memory bandwidth of our code, compared to that of the triad test in the Stream benchmark [27]. On one hand, these histograms underline that the update-velocities, accumulation and sorting steps are far to reach the peak memory bandwidth and thus, they have a good scaling up to 24 threads. On the other hand, the update-positions step reaches the same memory bandwidth as the Stream benchmark (the theoretical peak on 24 threads is 127.99 GB/s). Accordingly, this step cannot be further accelerated when using 24 threads. The strip-mining idea thus becomes natural when looking at those figures: when the update-positions loop cannot be further sped up by more threads, due to memory bandwidth limits, merging it with another loop becomes a good idea. To merge two loops, we can either "undo" part of our loop fission, by applying loop fusion [21, Section 9.2]; or we can use strip-mining. To preserve the high

efficiency of this vectorized loop, it is better to use strip-mining rather than loop fusion, which would break the vectorization opportunity.



Test case in Table 1.

Figure 13: Loop fission (3 loops) VS Strip-mining (2 loops).

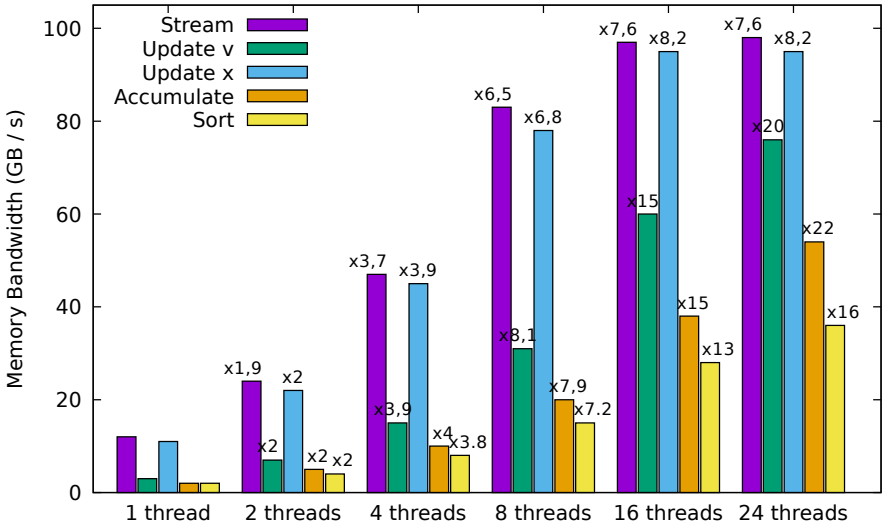
4.3. Weak scaling on 3,072 cores with MPI

Our parallel results come from simulations executed on the supercomputer Marconi. Each node has 2 sockets of 24 cores each, hence we used one MPI process per socket and 24 threads per process.

Fig. 16 shows a weak scaling from 1 core to 3,072 cores (64 nodes). These simulations run with 1 billion particles per socket in order to use the full memory. We can see that up to 3,072 cores, the overhead due to `MPI_ALLREDUCE` stays acceptable, thanks to the hybrid parallelism OpenMP + MPI.

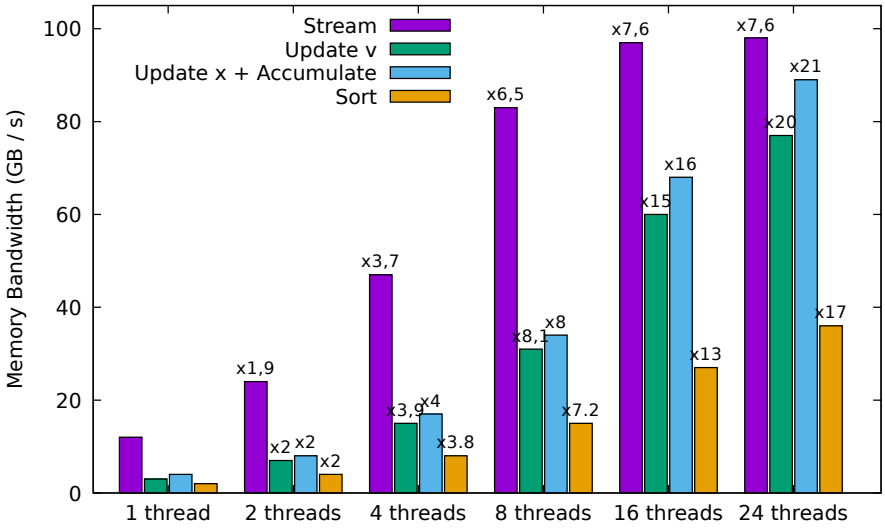
5. Conclusion and Outlook

In order to simulate kinetic plasmas, we developed a three-dimensional Particle-in-Cell code and studied its performance. We thus explored several



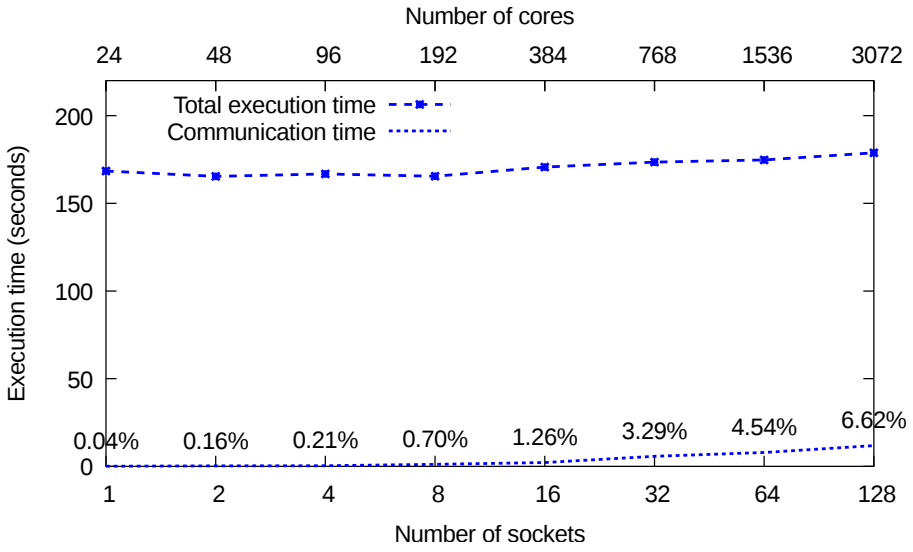
Test case in Table 1.

Figure 14: Memory Bandwidth of the loop-fission code (3 loops).



Test case in Table 1.

Figure 15: Memory Bandwidth of the strip-mined code (2 loops).



Test case: 64 x 64 x 64 grid, 1 billion particles per socket, 100 iterations simulation (sorting every 10 iterations). Architecture: Skylake. Communication time is also shown as percentage of the execution time.

Figure 16: Weak scaling on Marconi.

space-filling curves for the data layout of E and ρ, and different loop transformations for the particle loops, with the aim of improving the cache reuse and achieving efficient vectorization. More precisely, we introduced a novel space-filling curve in 3d, called L6D, which have been demonstrated to be the best strategy in the case of the Landau damping simulations. We compared our results to those of [9] and obtained significant gains when using at least a hundred particles per grid cell. We showed that strip-mining can be used in addition to the vectorization technique from [6] to further improve the code efficiency. We also implemented a hybrid MPI/OpenMP parallelism and we addressed memory bandwidth issues for justifying the scaling results with OpenMP.

In the future, it would be interesting to port our PIC code to Many Integrated Core (MIC) architectures to reinforce the multi-threading. Another challenging question would be to apply these optimization techniques to electromagnetic codes, in which the data structure analysis should be extended

when solving the Maxwell equations. The efficient PIC code we developed in this paper thus opens up the possibility to run realistic simulations in plasma physics in a three-dimensional physical space.

Acknowledgment

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom Research and Training Program 2014-2018 under Grant Agreement No. 633053. Simulations were run on the EUROfusion Marconi supercomputer, in the context of the Selavlas project led by K. Kormann. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

References

[1] Y. Barsamian, S. A. Hirstoaga, E. Violard, Efficient data structures for a hybrid parallel and vectorized particle-in-cell code, in: 2017 IEEE Intl. Parallel and Distributed Processing Symp. Workshops (IPDPSW), 2017, pp. 1168–1177. doi:10.1109/IPDPSW.2017.74.

[2] C. K. Birdsall, A. B. Langdon, Plasma Physics via Computer Simulation, McGraw-Hill, New York, 1985.

[3] R. W. Hockney, J. W. Eastwood, Computer Simulation Using Particles, Institute of Physics, Philadelphia, 1988. doi:10.1201/9781439822050.

[4] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, T. J. T. Kwan, Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation, Physics of Plasmas 15 (5) (2008) 055703. doi:10.1063/1.2840133.

[5] V. K. Decyk, S. R. Karmesin, A. de Boer, P. C. Liewer, Optimization of particle-in-cell codes on reduced instruction set computer processors, Computers in Physics 10 (3) (1996) 290–298. doi:10.1063/1.168571.

[6] H. Vincenti, M. Lobet, R. Lehe, R. Sasanka, J.-L. Vay, An efficient and portable SIMD algorithm for charge/current deposition in particle-in-cell codes, Comput. Phys. Commun. 210 (2016) 145–154. doi:10.1016/j.cpc.2016.08.023.

[7] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, A. Bhattacharjee, The plasma simulation code: A modern particle-in-cell code with patch-based load-balancing, J. Comput. Phys. 318 (2016) 305–326. doi:10.1016/j.jcp.2016.05.013.

[8] V. K. Decyk, T. V. Singh, Particle-in-cell algorithms for emerging computer architectures, Comput. Phys. Commun. 185 (3) (2014) 708–719. doi:10.1016/j.cpc.2013.10.013.

[9] A. Jocksch, F. Hariri, T.-M. Tran, S. Brunner, C. Gheller, L. Villard, A bucket sort algorithm for the particle-in-cell method on manycore architectures, in: Parallel Processing and Applied Mathematics: 11th Intl. Conf. (PPAM), 2016, pp. 43–52. doi:10.1007/978-3-319-32149-3_5.

[10] E. Chacon-Golcher, S. A. Hirstoaga, M. Lutz, Optimization of particle-in-cell simulations for Vlasov-Poisson system with strong magnetic field, ESAIM: Proceedings and Surveys 53 (2016) 177–190. doi:10.1051/proc/201653011.

[11] F. Panneton, P. L'Ecuyer, M. Matsumoto, Improved long-period generators based on linear recurrences modulo 2, ACM Transactions on Mathematical Software (TOMS) 32 (1) (2006) 1–16, (Source Code: http://www.iro.umontreal.ca/~panneton/WELLRNG.html). doi:10.1145/1132973.1132974.

[12] C. K. Birdsall, D. Fuss, Clouds-in-clouds, clouds-in-cells physics for many-body plasma simulation, J. Comput. Phys. 3 (1969) 494–511. doi:10.1006/jcph.1997.5723.

[13] K. J. Bowers, Speed optimal implementation of a fully relativistic particle push with charge conserving current accumulation on modern processors, in: Proceedings of the 18th Int. Conf. Numerical Simulation of Plasmas (ICNSP), 2003, pp. 383–386, http://web.mit.edu/ned/ICNSP/ICNSP_BookofAbstracts.pdf.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 2009.

[15] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, M. Thottethodi, Nonlinear array layouts for hierarchical memory systems, in: Proceedings of the 13th Intl. Conf. on Supercomputing (ICS), 1999, pp. 444–453. doi:10.1145/305138.305231.

[16] J. Mellor-Crummey, D. Whalley, K. Kennedy, Improving memory hierarchy performance for irregular applications using data and computation reorderings, Intl. Journal of Parallel Programming 29 (3) (2001) 217–247. doi:10.1023/A:1011119519789.

[17] D. DeFord, A. Kalyanaraman, Empirical analysis of space–filling curves for scientific computing applications, in: 42nd Intl. Conf. on Parallel Processing (ICPP), 2013, pp. 170–179. doi:10.1109/ICPP.2013.26.

[18] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, R. Widera, Radiative signatures of the relativistic Kelvin-Helmholtz instability, in: Proceedings of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC), 2013, pp. 5:1–5:12. doi:10.1145/2503210.2504564.

[19] I. Surmin, S. Bastrakov, Z. Matveev, E. Efimenko, A. Gonoskov, I. Meyerov, Co-design of a particle-in-cell plasma simulation code for Intel xeon phi: A first look at knights landing, in: Proceedings of the

16th Intl. Conf. on Algorithms and Architectures for Parallel Processing Collocated Workshops (ICA3PP, SCDT), 2016, pp. 319–329. doi:10.1007/978-3-319-49956-7_25.

[20] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216–231, http://www.fftw.org. doi:10.1109/JPROC.2004.840301.

[21] M. J. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, Tech. rep., IBM Ltd. https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39?OpenDocument (1966).

[23] R. Raman, D. S. Wise, Converting to and from dilated integers, IEEE Transactions on Computers 57 (4) (2008) 567–573. doi:10.1109/TC.2007.70814.

[24] D. Hilbert, Über die stetige abbildung einer linie auf ein flächenstück, Mathematische Annalen 38 (1891) 459–460.

[25] J. Skilling, Programming the Hilbert curve, Vol. 707, 2004, pp. 381–387. doi:10.1063/1.1751381.

[26] C. Severance, K. Dowd, High Performance Computing, OpenStax CNX, 2010, http://cnx.org/content/col11136/1.5/.

[27] J. D. McCalpin, Memory bandwidth and machine balance in current high performance computers, IEEE Computer Society Technical Committee on Computer Architecture Newsletter (TCCA) (1995) 19–25.