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

Yann Barsamian, Sever Adrian Hirstoaga, Eric Violard

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

Yann Barsamian
Inria Nancy-Grand Est, CAMUS Team
& Université de Strasbourg, CNRS,
ICube UMR 7357
F-67412 Illkirch, France
Email: ybarsamian@unistra.fr

Sever A. Hirstoaga
Inria Nancy-Grand Est, TONUS Team
& Université de Strasbourg, CNRS,
IRMA UMR 7501
F-67084 Strasbourg, France
Email: sever.hirstoaga@inria.fr

Éric Violard
Inria Nancy-Grand Est, CAMUS Team
& Université de Strasbourg, CNRS,
ICube UMR 7357
F-67412 Illkirch, France
Email: violard@unistra.fr

***Abstract*—The contribution of the present work relies on an innovative and judicious combination of several optimization techniques for achieving high performance when using automatic vectorization and hybrid MPI/OpenMP parallelism in a Particle-in-Cell (PIC) code. The domain of application is plasma physics: the code simulates 2d2v Vlasov-Poisson systems on Cartesian grids with periodic boundary conditions. Overall, our code processes 65 million particles/second per core on Intel Haswell (without hyper-threading) and achieves a good weak scaling up to 0.4 trillion particles on 8,192 cores.**

The optimizations mainly consist in using (i) a structure of arrays for the particles, (ii) an efficient data structure for the electric field and the charge density, and (iii) an appropriate code for automatic vectorization of the charge accumulation and of the positions' update. In particular, we use space-filling curves to enhance data locality while enabling vectorization: starting from a redundant cell-based data structure for the electric field and for the charge density, we compare several space-filling curves for an efficient ordering of these data and we obtain a gain of 36% in the number of L2 and L3 cache misses when using a Morton curve instead of the classical row-major one. In addition, by proposing a specific writing of the updating positions code we achieve a 31% time improvement in that step. The optimizations bring an overall gain in the execution time of 42% with respect to a standard code.

The parallelization of the particle loops is simply performed by means of both distributed and shared memory paradigms, without domain decomposition. We explain the weak and the strong scalings of the code bounded as expected by the overhead of the MPI communications.

***Keywords*-data structures; space-filling curves; SIMD architecture; hybrid parallelism; strong and weak scaling; Particle-in-Cell simulation; plasma physics**

I. Introduction

The Particle-in-Cell (PIC) method is a practicable tool for simulating a wide range of phenomena in plasma physics. Although their basic form is easy to implement, PIC codes require very large computing resources, for at least two reasons: one challenge appears when dealing with problems entailing multi-scale behavior, since the numerical parameters are to be chosen very small for solving the smallest scale in time and/or space; the other issue stems from a complex implementation of the data structures and the subsequent required communications, needed for achieving simulations with reasonable time execution.

PIC algorithms are used to simulate plasmas by integrating self-consistently the trajectories of charged particles with fields that are generated by the particles themselves [1, 13]. In the case where there is no other external field and the self-induced magnetic field is neglected, this relies on solving the following Vlasov-Poisson system:

$$\begin{cases} \partial_t f + v \cdot \nabla_x f + \dfrac{q}{m} E \cdot \nabla_v f = 0 & \text{Vlasov} \\ -\Delta \phi = \dfrac{\rho}{\epsilon_0} & \text{Poisson} \end{cases}$$

where

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

In the system above, $f = f(x,v,t)$ stands for the distribution of one species of particles (with charge q and mass m) in a four-dimensional phase space (two dimensions for positions and two dimensions for velocities), ρ stands for the charge density, and E for the self-induced 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. A simulation is thus defined by the time stepping for solving the characteristics equation, the discretization for solving the Poisson equation, and the interpolation/accumulation schemes. More precisely, the loop to be applied at each iteration 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 PIC codes we thus have two types of data, particles and grid quantities (E and ρ) that need to communicate one with another. Usually, the most time consuming steps are: first, the accumulation and interpolation steps which are dominated by data motion and thus, avoiding waste of resources requires appropriate data structure for improving data locality; second, the updating-positions step which is dominated by computations. Thus, the following two ideas are promising ways to explore in order to enhance performance in a PIC simulation: first, propose data structures for

E and ρ that are suitable for minimizing the cache misses and second, use vectorization to accelerate the execution of the updating-positions and the charge accumulation steps.

This paper focuses on sequential and parallel implementation, in `C`, of a minimal two-dimensional (2d) electrostatic PIC code for simulating kinetic plasmas. Our first goal is to combine several single core optimizations, in order to improve cache reuse and to achieve good vectorization on Single Instruction Multiple Data (SIMD) architectures (e.g. AVX2-256 bits data registers). Then, our second aim is to parallelize this optimized code on thousands of cores by using multi-processing and multi-threading.

The remainder of this paper is organized as follows: in Section II we detail the steps of the PIC implementation and introduce the related work. In Section III we explain our contributions. In Section IV we detail the code optimizations and we present the performance results on single core. In Section V we show the scalability of the code on up to 8,192 cores of the supercomputer Curie. Section VI summarizes the work and presents some future directions.

```
Parameters:
$N$ the number of particles.
$\Delta t$ the time step.
$ncx \times ncy$ the number of cells of the spatial grid.

Variables:
$particles[N]$ the particles.
$\rho[ncx \times ncy]$ the charge density.
$E[ncx \times ncy]$ the self-consistent electric field.

Initialization:
1   Initialize $particles$ with $N$ particles and sort it
2   Compute $\rho$ and $E$ at $t = 0$

Algorithm:
3   **Foreach** time iteration, **do**
4       **If** $(condition)$, **then**
5           Sort the particles
6       **End If**
7       Set all cells of $\rho$ to 0
8       **Foreach** particle in $particles$,
9           Update the velocity                      $v{+}= \frac{q}{m}\Delta t\, E$
10          Update the position                      $x{+}= \Delta t\, v$
11          Accumulate the charge on the grid points
12      **End Foreach**
13      Compute $E$ from $\rho$                       Poisson solver
14  **End Foreach**
```

Figure 1. Particle-in-Cell pseudo-code.

II. Implementation Steps and Related Work

In our PIC code, the particle positions and velocities are initialized randomly, each with weight w. They are advanced in time with a leap-frog time stepping. 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 (first-order weighting), meaning that four 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.

Each particle is represented as a combination of a cell index and normalized offsets within this cell. The advantages of this representation are exposed in [3, Section III-E]. In addition to the parameters explained in Fig. 1, we denote the physical space by $[x_{min}; x_{max}) \times [y_{min}; y_{max})$, and the grid spacing by $\Delta x = (x_{max} - x_{min})/ncx$ and $\Delta y = (y_{max} - y_{min})/ncy$. Thus, a particle positioned at $(x_{\text{physical}}, y_{\text{physical}})$ is mapped on the grid at the position $(x, y) \in [0; ncx) \times [0; ncy)$, where

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

Then, we consider the integers

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

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

$$\texttt{dx} = x - i_x \quad \text{and} \quad \texttt{dy} = y - i_y.$$

The cell index i_{cell} in $\{0, 1, \ldots, ncx \times ncy - 1\}$ is the image of some one-to-one mapping depending on (i_x, i_y). Commonly in `C`, the row-major mapping is used:

$$(i_x, i_y) \mapsto i_{cell} = i_x \times ncy + i_y.$$

The standard 2d representation of the electric field E and of the charge density ρ stores their values at the grid points:

```
double Ex[ncx][ncy];
double Ey[ncx][ncy];
double rho[ncx][ncy];
```

In this case, the interpolation of the 2d arrays `Ex` and `Ey` asks for accessing memory locations that are not contiguous. A solution to partially overcome this problem consists in storing components of the field in only one array [7, Section IV, Case 3]:

```
double Exy[ncx][ncy][2];
```

Unfortunately, this data structure still leads to non-contiguous accesses. This problem is solved by using a redundant one-dimensional array of coefficients [4, 2]:

```
double E_1d[ncx*ncy][8];
```

This redundant array `E_1d` stores, for each cell, values of each of the two field arrays at the grid points on the four corners of the cell, contiguously in memory. It takes four times more memory than the standard layout, but we will demonstrate how to manage it in order to gain overall performance through additional cache hit improvements.

This redundant structure, applied for the charge density ρ, recently opened the possibility to vectorize the accumulation step (line 11 in Fig. 1) [20, Section 4.1.2.]. Differences between standard and redundant arrays are recalled in Fig. 2.

```
double rho[ncx][ncy]; // Standard 2d.
rho[i_x  ][i_y  ]+=w*(1-dx[i])*(1-dy[i]);
rho[i_x  ][i_y+1]+=w*(1-dx[i])*(  dy[i]);
rho[i_x+1][i_y  ]+=w*(  dx[i])*(1-dy[i]);
rho[i_x+1][i_y+1]+=w*(  dx[i])*(  dy[i]);
double rho_1d[ncx*ncy][4]; // Redundant.
float cx[4] = {  1.,  1.,  0.,  0.};
float sx[4] = { -1., -1.,  1.,  1.};
float cy[4] = {  1.,  0.,  1.,  0.};
float sy[4] = { -1.,  1., -1.,  1.};
for (corner = 0; corner < 4; corner++)
  rho_1d[i_cell[i]][corner] += w *
    (cx[corner] + sx[corner] * dx[i]) *
    (cy[corner] + sy[corner] * dy[i]);
```

Figure 2. Accumulate: Standard VS Redundant.

However, in that paper, this data structure is used for vectorization without discussing cache improvement.

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 4-6 in Fig. 1) [7, Section VI]. The cost of this step is made linear in N by using the bucket sorting, since the number of cells is much smaller than that of the 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 [5], or with irregular accesses such as the n-body problem [15]. Nevertheless, none of those 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.

The space-filling curves are also of interest in particle codes at the inter-process level, to achieve load balancing when using domain decomposition [11] or to minimize communication between processes [8], which has no impact on cache performances.

III. Contributions

We address the following issues:

- ⋆ Study the behavior of space-filling curves in the context of optimizing the cache performance of a PIC code.
- ⋆ Combine many optimization techniques that exist, in order to maximize their associated advantages and to minimize their associated drawbacks.
- ⋆ Evaluate the performance of a hybrid MPI/OpenMP parallelism by discussing the scaling of the code in tandem with the memory channels and bandwidth.

A first contribution in the optimization of the code on single core relates to the redundant cell-based structure for E and ρ. We will discuss in Section IV-B how to gain substantial performance by choosing a suitable mapping $(i_x; i_y) \mapsto i_{cell}$, while still preserving vectorization. To the best of our knowledge, no explicit results over the update-velocities and accumulation steps were shown before. More precisely, we show that the L4D curve [5], which is usually not studied because it is not recursive, has the best properties for our code. Recent results [20] targeting the vectorization of the accumulation step are thus shown to be enhanced with the technique we propose.

A second contribution of this work is to show how to achieve efficiency when using automatic vectorization, via loop transformation and code rewriting. More specifically, the loop transformation aims at separating the steps of the PIC simulation in different particle loops instead of treating all the steps in one single particle loop. This approach allows us to customize the vectorization of the updating-positions step, using a bitwise operation which clearly reduces the computation time. We need to notice that this handling is possible since periodic boundary conditions are used. It is shown that the extra memory movement necessary for this splitting is more than compensated by the vectorization.

Our last contribution concerns the parallelism. We will first discuss multi-threading parallelism (with OpenMP) on single node, showing, as expected, the boundedness of the strong scaling by the number of the memory channels. We also detail the memory bandwidth numbers within different loops of the PIC algorithm and we compare them to the reachable peak. We then discuss the parallelism with multiple processes (using MPI). The typical approach is to use domain decomposition. Unlike this method, we simply assign different lists of particles to different processes. The only communications between processes arise when computing ρ as the sum of the charge densities associated with the particles of each process. We demonstrate that as long as we use the full memory of every core, we obtain a good weak scaling up to 8,192 cores in a half-trillion particles simulation.

IV. Single Core Optimizations

We present in this section gains in performance achieved by optimizing a previous work [4]. The hardware performance counters were tracked with `perf` and PAPI [19].

All the simulations in this paper ran on:

- ⋆ the local computing machine “Icps”: 2× (Intel Xeon E5-2650 v3 @2.3 GHz (Haswell), 32 GB of RAM, 2 memory channels, 10 cores).
  Compilers: GNU 6.2 and Intel 17.0.0
- ⋆ the supercomputer “Curie”[1]: 5,040 nodes, each node has 2× (Intel Xeon E5-2680 @2.7 GHz (SandyBridge), 64 GB of RAM, 4 memory channels, 8 cores).
  Compilers: GNU 6.1 and Intel 16.0.3.210.

In addition to the optimization flag `-O3`, when using the GNU compiler we added `--param max-completely-peeled-insns=0` since otherwise, the accumulation loop in Fig. 2 gets completely peeled,

[1]http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm

thus preventing vectorization. We used the library FFTW3 [9] for the Poisson solver.

The results shown in this section were all obtained with the test case presented in Table I. In addition, we also simulated nonlinear Landau damping and two-stream instability test cases. Theoretical results which allow to verify the code are available [1, 13]. Thus, we checked the numerical conservation of the total energy and the numerical evolution in time of the electric field.

Table I
TEST CASE FOR SEQUENTIAL OPTIMIZATIONS.

Physical test case	Linear Landau damping
Spatial grid size	128 x 128
Number of particles	50 million
Number of iterations	100 (sorting every 20 iterations)
Architecture	Single core on Haswell

A. Loop splitting

The first optimization we implemented is the loop splitting. More precisely, the loop on lines 8-12 in Fig. 1 is broken into three parts: one loop to update-velocities (line 9), one to update-positions (line 10), and one loop to accumulate the charge (line 11). 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) we obtain an 18% to 25% gain in performance without vectorization, depending on the data structure. Indeed, even if we have to scan the particle array three times, a separate processing of the arrays of E and ρ in different loops leads to a better overall memory management.

B. Data structure for E and ρ

As already mentioned, we have at our disposal two data structures for E and ρ: the standard 2d and the redundant one. The latter has been shown in [20, Section 4.1.2.] to be effective for SIMD architectures since it enables vectorization of the accumulate loop.

We next show that using other memory layouts for the redundant data structure decreases the number of cache misses. We already emphasized the fact that in PIC codes, memory accesses are a major bottleneck. Every time the code accesses a cell of E or ρ, a contiguous portion of that array is loaded into the cache: we want to do all the computations that use these data cells while they are still there, avoiding to reload them later from the main memory.

Since particles are moving at each iteration, a periodic sorting of the particles needs to be applied in order to improve data locality. In this manner, two particles contiguous in memory are in the same grid cell and thus, they access the same E (or ρ) cell during the update-velocities (or accumulate) loop. 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, using the previous notations, our aim is to find a mapping $(i_x; i_y) \mapsto i_{cell}$ so that we obtain, when a particle changes a cell, a high probability that its new cell-index i_{cell} is close to the old one.

It should be noted that the mapping $(i_x; i_y) \mapsto i_{cell} = i_x \times ncy + i_y$ (row-major ordering) has a good data locality when a particle moves along the y-axis: if i_y increases by one, the new cell-index also increases by one (except for particles on the right edge of the grid), becoming exactly the index accessed by the following particles in the particle array. However, when a particle moves along the x-axis, this good behavior is lost: if i_x increases by one, the cell-index changes by ncy 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 $(i_x; i_y) \mapsto i_{cell}$:

(i) Scan-order (or row-major order): the canonical `C` memory layout.
(ii) "Column-major of row-major"-order (or L4D-order), cf. [5, Section 2.1.] and Fig. 4. We re-designed algorithms to convert from and to this ordering.
(iii) Morton-order (И-order) or Lebesgue-order (Z-order), cf. [16] and Fig. 3. Algorithms to convert from and to this ordering can be found in [17].
(iv) Hilbert-order, cf. [12]. Algorithms to convert from and to this ordering can be found in [18].

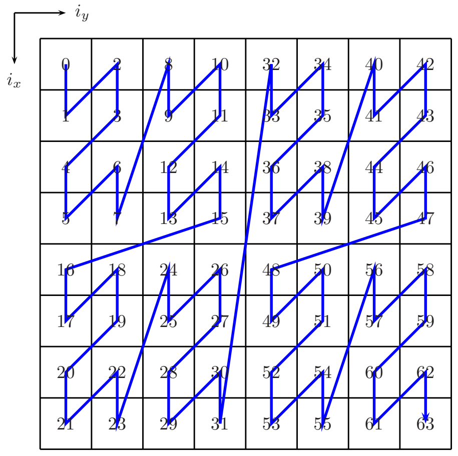


Figure 3. Morton layout of a 8 x 8 matrix.

We present in Fig. 5 and Fig. 6 the evolution in time of the number of cache misses for each of these orderings, for the L2 and L3 cache levels. As for the L1 level, we obtain very close values for all the orderings, see also Table II. At the first iteration, particles are sorted according to the given ordering. Then, we remark the steep descent of the cache misses number every 20 iterations due to the sorting.

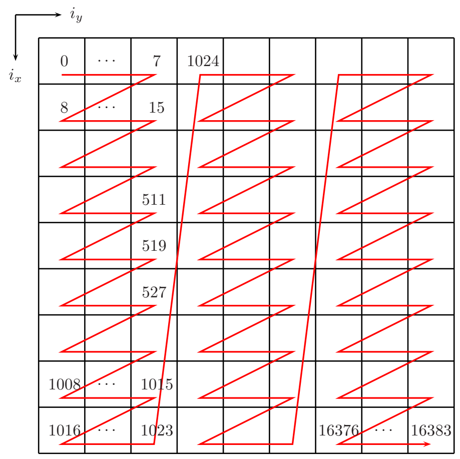


Figure 4. L4D layout of a 128 x 128 matrix, SIZE=8.

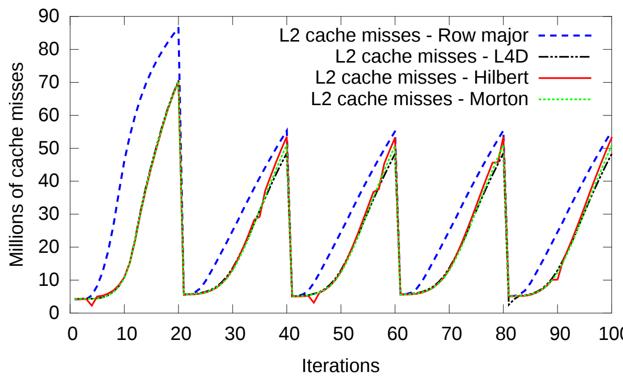


Figure 5. Millions of cache misses per iteration for the cache level 2 during the update-velocities and accumulate loops. Test case in Table I.

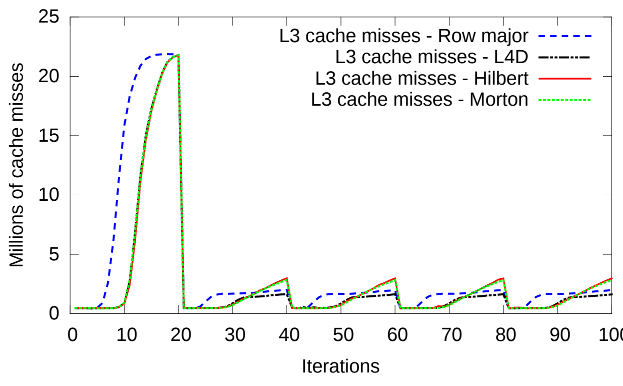


Figure 6. Millions of cache misses per iteration for the cache level 3 during the update-velocities and accumulate loops. Test case in Table I.

Clearly, the general idea those figures underline is that the three non-canonical layouts entail less cache misses than the row-major one. Going further, we see that all the curves give similar results when the particles become randomized (at least for the L2 cache). However, the good data locality

Table II
MILLIONS OF CACHE MISSES PER ITERATION, ON AVERAGE.

Ordering \ Cache level	L1	L2	L3
Row-major	95.4	43.3	4.94
L4D	92.0	27.8	3.14
И-Morton	91.1	27.0	3.20
Hilbert	90.9	27.1	3.29
Improvement (w.r.t. row-major)	−3.5%	−36%	−36%

Update-velocities and accumulate loops. Test case in Table I.

due to the sorting keeps longer in time for the three non-canonical curves than for the row-major.

As a drawback, we notice that the time to compute i_{cell} to/from i_x and i_y is greater for these layouts than for the row-major one. Thus, we need to compare in the following the overall performance of these space-filling curves. The L4D and Morton curves give the best results overall as reported in Table III. The update-positions loop takes much larger times when using the Hilbert ordering, the cause being that, to the best of our knowledge, there is no “efficient-enough” algorithm for computing this bijection. Therefore being the slowest in overall simulation time, the Hilbert ordering has to be discarded. We note that the total column is obtained by adding to the sum of the three first columns the time of sorting. For the L4D-order, we have to choose carefully the SIZE number depending of the cache sizes. In our tests, SIZE=8 led to the best times, thus in Fig. 4, we also chose SIZE=8, a divisor of 128. It is to note that choosing a value of SIZE that does not divide ncy is possible: then, there will be a few allocated cells that correspond to physical positions outside the boundaries and that will never be accessed. The Morton-ordering gives the same speedup as the L4D-ordering, and does not depend on cache sizes: the update-velocities and accumulate loops become cache-oblivious [10]. Those orderings are faster than the row-major ordering for the accumulate loop (a 15% gain using Intel and an 11% gain using GNU), that was already better than the standard 2d structure thanks to vectorization. The redundant data structure with row-major ordering is not better than the standard 2d structure for the update-velocities (5% lost with Intel, no time change with GNU), but with those orderings, we are able to gain time (3% gained with Intel, 9% gained with GNU). This might not seem significant but we emphasize that the new data structure needs four times more memory than the standard 2d one.

The results in Table III show 3 extra seconds in the update-positions loop for the L4D and Morton layouts. The reason is that for these two layouts we store, in addition to the cell-index i_{cell}, the indexes i_x and i_y for each particle. We also tested the computation of i_x and i_y from i_{cell}, instead of storing them and we remarked that this is a much slower approach. In contrast, for the row-major layout, that computation can be done in only one operation and therefore

Table III
TIME SPENT IN THE DIFFERENT LOOPS (IN SECONDS).

	Update v	Update x	Accumulate	Total
2d standard	30.6	12.5	20.7	74.3
Row-major	32.3	12.8	14.9	70.5
L4D	29.7	15.9	12.7	68.8
Morton	29.6	15.3	12.7	69.0
Hilbert	30.0	133.1	12.8	185.8

Test case in Table I.

we do not need to store i_x and i_y.

The computation of i_{cell} can be achieved via different algorithms. In [17], two algorithms for the Morton layout are proposed: one that takes 12 operations and one that takes 5 operations plus two loads from a lookup table. We implemented the same idea for the L4D layout. In both cases, the lookup table creates an indirection which is not vectorizable and therefore this approach has to be discarded. Thus, we chose the Algorithm 5 from [17] for the Morton layout and we propose the following one for the L4D-order:

$$\begin{aligned}(i_x; i_y) \mapsto \text{SIZE} \times i_x + mod(i_y, \text{SIZE}) \\ + ncx \times \text{SIZE} \times (i_y/\text{SIZE}).\end{aligned}$$

Fig. 4 allows us to explain why the L4D-order makes less cache misses than the row-major order. In the following explanation, we chose SIZE=8 as in the picture. It can be replaced with other values, as long as they are not too large for the cache (notice that SIZE=ncy corresponds to the row-major ordering). In this context, when a particle moves horizontally, $\frac{7}{8}$ of the time, it will lead to a new index close to the old one (i_{cell} changed to $i_{cell}+1$); only $\frac{1}{8}$ of them will thus generate cache misses twice. All the vertical moves lead to a new index close to the old one (i_{cell} changed to $i_{cell}+8$) - except on the boundary. Contrast this with the row-major ordering, in which all the horizontal moves are good (i_{cell} changed to $i_{cell}+1$) and all the vertical ones are bad (i_{cell} changed to $i_{cell}+128$); assuming that the particles move along the x and y-axes with no favorite direction, it means that we can expect up to $43\%\big(=\frac{7}{8}\times 50\%\big)$ less cache misses on E and ρ. The overall improvement shown in Table II is nevertheless lower since we have to take into account cache misses from the particle array.

C. *Vectorization of the update-positions loop*

SIMD architectures can handle several operations at once: they compute on vectors rather than on scalars. For example, with vectors of size 256 bits, it takes as much time to compute 4 multiplications on double-precision real numbers (each of size 64 bits) than to compute only one. The `-ftree-vectorize` compilation flag (activated from `-O2` with Intel, from `-O3` with GNU) is one possibility to automatically vectorize the code. However, in order to enable real vector performances we need to rewrite the code in addition to the use of an appropriate data structure.

1) Array of Structures or Structure of Arrays?: To achieve the full power of vectorization requires that the Single Instruction operates on Multiple Data that are contiguous in memory. Using array of structures (AoS) for the particles leads to stride of 4 or 8 between two data to be vectorized. The GNU compiler does not vectorize such a code while using the Intel compiler leads to unsatisfactory timings. Thus, using a structure of arrays (SoA) guarantees the best timing for the update-positions loop: a 23% gain with Intel and a 30% gain with GNU.

2) Remove the `ifs`: When updating the positions of the particles in a periodic setting, we need to get in the physical domain the particles going outside. Usually, this is done by testing if the new position is still inside the grid. Without any concern for vectorization, `ifs` in a program are expensive (when incorrectly predicted, they cause rollbacks and inhibit the filling of arithmetic pipelines). Moreover, they prevent automatic vectorization (for the GNU compiler) or at best give unsatisfactory vector code. Therefore, our goal is to remove the `ifs` thanks to an efficient rewriting of the code.

As shown in Section II, the positions of the particles are stored each with an integer (for the nearest lower grid position) and a real number (for the distance to that grid position). Since periodic boundary conditions are used, if a particle leaves the grid from one side, it goes to the beginning of the grid from the opposite side. This can be implemented by using an extension of the modulo[2] over the reals: `modulo(a, b) = a - floor(a / b) * b)`:

```
if (x < 0. || x >= ncx)
    x = modulo(x, ncx);
i_x = floor(x);
dx[i] = x - i_x;
```

Two ideas for removing the `ifs` are proposed in [7]. They both suppose that a particle will not move away further than one cell from the grid. If this can be acceptable in many simulations, we must take into account the general case (hence, the need for modulo when a particle crosses more than one cell). Therefore, we propose the following idea: if we have a fast way to compute the modulo, then we can avoid testing if the particle moves away from the grid; we can compute the modulo for each particle and it will be faster. We consider the following computation

```
i_x = modulo(floor(x), ncx);
dx[i] = x - floor(x);
```

This second formulation is faster, because it is a modulo over the integers where the divisor is a power of two. In that case, computing `modulo(a, b)` is equivalent to computing a bitwise and between `a` and `b - 1` (because the numbers are encoded in binary). For example, if we

[2]$modulo(a,b)$ is the unique real number in $[0;b)$ such that $a - modulo(a,\ b)$ is an integer multiple of b.

have 128 grid cells we have to compute modulo 128 and computing $modulo(a, 128)$ in binary is the same as taking the seven least significant bits; in other words, it is exactly a bitwise and between a and $127_{10} = 1111111_2$[3]. Since it is well-known that bit operations are extremely fast, this approach reduces the time of the update-positions loop.

3) Remove function calls: In the previous piece of code one can see a call to the `floor` function. The Intel compiler vectorizes such a function call, but this is not the case for the latest GNU compiler. In order to solve this problem, we can rewrite the code as follows:

```
floor_x = (int)x - (x < 0.)
i_x = floor_x & ncx_minus_one;
dx[i] = x - floor_x;
```

The first line computes `floor(x)` because the cast to an integer removes what is after the comma: if the number is negative, we have to remove 1 (which is done by the return of the test in C, 1 for `true` or 0 for `false`).

Rewriting the code in this way enables automatic vectorization by the GNU compiler too. In addition, the Intel compiler produces faster vectorized code (31% time improvement on the update-positions loop).

D. Loop hoisting

We also improve runtime performance by removing as much computations as possible from the particle loops. Every computation that needs a constant can be done outside the loops. For example: (a) to update the velocities of particles, we need to multiply the field by $\frac{\Delta t \times q}{m}$ and (b) to update the positions of particles, we need to multiply the velocities by $\frac{\Delta t}{\Delta x}$. All these multiplications can be done outside the particle loop if, instead of storing the field and the velocities, we store the field multiplied by $\frac{\Delta t^2 \times q}{m \times \Delta x}$ and the velocities multiplied by $\frac{\Delta t}{\Delta x}$. This optimization leads to a gain in time of 3.5% with Intel and of 2% with GNU.

E. Overall gains and comparisons

In this section, we presented sequential optimizations. When using all the optimizations together, we remark that the code runs slightly faster with the Intel compiler than with the GNU one (1.8%). Our optimizations are thus summarized in Table IV with the former. In this table, the baseline is a version of the code with the standard 2d data structure for E and ρ and the Array of Structures for the particles. The gains (in %) are computed with respect to the previous line and the accumulated gains are computed with respect to the baseline.

- ⋆ Loop hoisting: $\mathcal{O}(ncx \times ncy)$ operations instead of $\mathcal{O}(num\_particles)$ operations.
- ⋆ Loop splitting: better memory management, allows to vectorize the update-positions loop.
- ⋆ Redundant arrays: vectorized accumulation loop.
- ⋆ Structure of Arrays: stride-1 vectorization in the update-positions loop.
- ⋆ Space-filling curves on the redundant data structure: 36% less cache misses on the accumulate and update-velocities loops.
- ⋆ Optimized update-positions loop (no control flow, no function call): 31% less time on that loop, able to vectorize with the GNU compiler too.

[3]It works also for negative numbers, thanks to the two's complement.

Table IV
TOTAL EXECUTION TIME, GAINS AND ACCUMULATED GAINS.

	Time(s)	Gains(%)	Acc. gains(%)
Baseline	120.4	0.0	0.0
+ Loop Hoisting	113.4	5.8	5.8
+ Loop Splitting	97.9	13.7	18.7
+ Redundant arrays (E and ρ)	94.0	4.0	21.9
+ Structure of Arrays ($particles$)	76.0	19.1	36.9
+ Space-filling curves (E and ρ)	72.6	4.5	39.7
+ Optimized update-positions loop	68.8	5.2	42.8

Test case in Table I.

Overall, these optimizations result in 75 million particles processed per second, on one hyper-threaded core on Intel Haswell architecture (65 million particles processed per second without hyper-threading). Those performances are compared in Table V to the electrostatic 2d Vlasov-Poisson code presented in [6], which uses domain-decomposition. The loop called "push" in that paper corresponds to the two loops update-velocities and update-positions in our code. After that push, some of the particles move from one subdomain to another. This is treated in a step called "reorder" which is not a full sorting, like in our algorithm. The reorder from that paper and the sorting in our work are separated in Table V because they are not directly comparable. For the sorting, we ran several simulations and we found that the optimal number of iterations between two sorting steps is 50 on Sandy Bridge architecture, and 20 on Haswell architecture. The results in Table V are thus presented for these sorting frequencies. Those experiments underline that the optimal number of iterations between two sorting steps can vary according to the architecture. Therefore it will be interesting to implement an automatic finding of this optimal number. This is left for future work.

V. THREAD-LEVEL AND PROCESS-LEVEL PARALLELISM

In our code, we used the same kind of parallelism as in [4]. The new features from our code compared to that paper are the parallelization of the sorting among threads and OpenMP 4.5 reduction on array sections.

A. 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 subdomains and to assign

Table V
TIME SPENT PER PARTICLE PER ITERATION, IN NANOSECONDS.

	Decyk & Singh work [6] (on Nehalem)	Present work (on Sandy Bridge)	Present work (on Haswell)
Push	19.9	15.6	9.1
Accumulate	9.0	4.3	2.6
Reorder	0.3	–	–
Sorting	–	1.9	2.0
Total	29.2	21.8	13.7

Architecture : single core, processor given in each column.

the particles inside a subdomain to a processor (among the wide literature, see e.g. [3, 11]). In grid-based simulations, this method was established to give good scaling, as long as the work due to the communications through the subdomain boundaries remains small compared to the computations inside the subdomains. However, the main drawback of this technique is the difficulty of maintaining the load balance. In this work, we handle the process-level parallelism without 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 advantages of this method are:

- ⋆ its simplicity: everything is automatically work-balanced, because every MPI process keeps all its particles during the whole simulation;
- ⋆ 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.
- ⋆ the scaling is automatically independent of the particle distribution and thus of the particle dynamics. Therefore we expect the performance of the parallelism to be problem independent.

The bottleneck of this approach is that the scalability is highly limited by the global reduction step. Thus, two parameters should not be very large, otherwise they could severely slow down the simulation: the number of cells and the number of processes. The aim of this section is to determine how many cores (processes) can be used until the MPI communications become costly, in the case when the number of cells is fixed to a few hundreds per dimension.

B. Thread-level parallelism

On modern architectures, several cores share common memory locations (starting from the L3 cache in our test computers). Thus, using thread-level parallelism has two benefits: (a) it enables better memory management and (b) there is a low communication overhead. Both of those assertions can be verified on the results which are reported in the next section.

1) Parallel sorting of particles: We compared an in-place algorithm for sorting with an out-of-place one (that uses an additional array of particles) and we found that the out-of-place one is twice as fast. This can be explained as follows: for every particle, the in-place algorithm needs three operations to change its position in the array, whereas the out-of-place needs only one. Nevertheless, because we need twice as much memory with this kind of sorting, we can put only half as much particles on each processor. Thus, we need twice as much processors for the same simulation.

This out-of-place algorithm has another major feature: it can be easily made parallel among threads. Remember that we use a counting sort: to make it parallel, we just have to give a set of cells to manage to every thread. Each thread will then only process particles that are in the cells it manages.

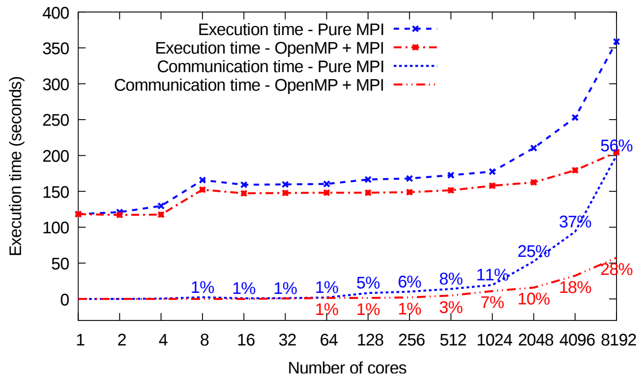
2) Array section from OpenMP 4.5: The update-velocities and update-positions loops are easily made parallel with `#pragma omp for`. The only problem arises for the accumulate loop. Only using the `#pragma omp for`, we have race conditions: particles from different threads will update the same ρ values. OpenMP 4.5 can handle this by adding `reduction(+:rho[0:ncx*ncy][0:4])` to the pragma. Nevertheless, this OpenMP 4.5 feature was not available with the latest Intel compiler. To exploit the previous advantages from this compiler, we rewrote this feature by hand (our hand-coded version showed no overhead with gcc 6.2 when compared to the OpenMP 4.5 feature).

C. Parallel results

Our parallel results come from simulations executed on the supercomputer Curie. Each node has 2 sockets of 8 cores each, hence for the hybrid MPI/OpenMP results, we used one MPI process per socket and 8 threads per process. For the pure MPI results, we used one MPI process per core.

Fig. 7 shows a weak scaling from 1 core to 8,192 cores (512 nodes, 10% of the total number of nodes of the Curie supercomputer). These simulations run with 50 million particles per core in order to use the full memory of each core. We can see that up to 8,192 cores, the overhead due to `MPI_ALLREDUCE` stays acceptable with the hybrid parallelism, whereas it becomes a major bottleneck when using only MPI. The hybrid parallelization achieves 543 million particles processed per second per node (2×8 threads).

Table VI shows a strong scaling up to 8 cores, when using 50 million particles (maximum memory on 1 core) over one socket. This table and Fig. 7 illustrate that our code reaches near-ideal scalability up to 4 threads, but not for 8 threads. The reason is that a PIC code is bounded by memory accesses and therefore, the 4 memory channels per socket limit the scalability when using more than 4 threads. To go further, we show in Fig. 8 the memory bandwidth of our code, compared to that of the triad test in the Stream



Test case: 128 x 128 grid, 50 million particles per core, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge. Communication time is also shown as percentage of the execution time.

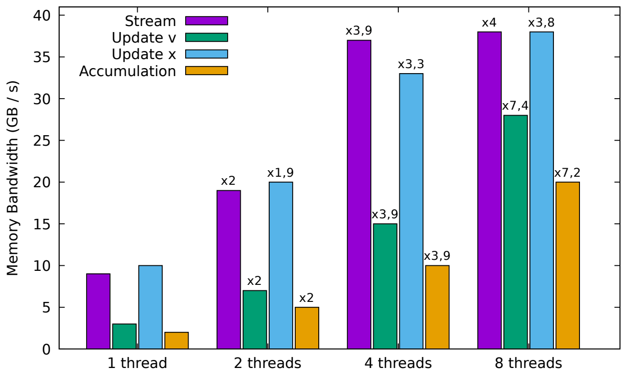
Figure 7. Weak scaling on Curie : Hybrid VS Pure MPI.

benchmark [14]. On one hand, this histogram underlines that the update-velocities and accumulation steps are far to reach the peak memory bandwidth and thus, they have a good scaling up to 8 threads. Their low memory bandwidth is explained by the high number of cache misses on the E and ρ arrays, despite the use of space-filling curves. On the other hand, the update-positions step reaches the same memory bandwidth as the Stream benchmark (the theoretical peak on 8 threads is 51.2 GB/s). Accordingly, this step cannot be further fastened when using 8 threads.

Table VI
STRONG SCALING ON ONE SOCKET OF CURIE (PURE OPENMP).

Number of cores	1 core	2 cores	4 cores	8 cores
Million particles/s	45.8	89.9	170	266
Million particles/s - ideal	45.8	91.6	183	366

Test case: 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge.



Test case: 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge.

Figure 8. Memory Bandwidth on one socket of Curie (Pure OpenMP).

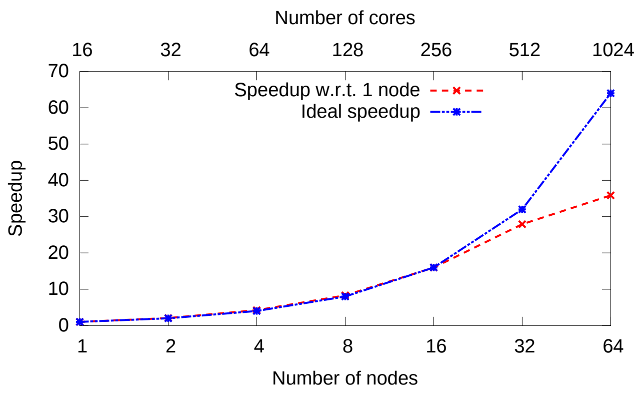
In consideration of the previous comments, it is not straightforward to extrapolate to 8 threads the overall gain results presented in Section IV-E. On 1 thread, we demonstrated that splitting the loops over the particles coupled with the SoA layout was the best choice even if frequent memory movement occurs. In Table VII we show that using the SoA layout with 3 loops is still the best option on 8 threads.

Table VII
TIME SPENT IN THE SIMULATION ON 8 THREADS (PURE OPENMP).

AoS, 1 loop	AoS, 3 loops	SoA, 1 loop	SoA, 3 loops
30.9 s	22.7 s	23.1 s	18.3 s

Test case: 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge.

Fig. 9 shows the strong scaling of the hybrid parallelism when using 800 million particles (maximum memory on one node). The last timing on 1,024 cores is less than 5 seconds. We thus remark that the speedup is far from ideal, when running over 64 nodes (128 processes). This is an expected result: the communication time as percentage of the total time grows with the increasing processes number, while the computation time per process decreases (since the number of particles per process decreases). Thus, in the case of the 64 nodes, only 6.25 million particles are distributed per process and the MPI communications take 32% of the total time. Going back to Fig. 7, when using 400 million particles per process, the same number of processes leads to far better results (constant weak scaling) since communications take in this case only 7% of the total time.



Test case: 256 x 256 grid, 800 million particles, 100 iterations simulation (sorting every 20 iterations). Architecture: Sandy Bridge.

Figure 9. Strong scaling on Curie (Hybrid: OpenMP + MPI).

VI. CONCLUSION AND FUTURE OUTLOOK

We developed an efficient PIC code for simulating kinetic plasmas. We explored several space-filling curves for the data layout of E and ρ and AoS vs. SoA layouts for particles with the aim of improving the cache use and achieving efficient vectorization. We compared our results to those of [6] and [20] and we obtained significant gains.

We implemented a hybrid MPI/OpenMP parallelism and we addressed memory bandwidth issues for justifying the scaling up to 8 threads. Then, going further to thousands of cores allowed us to measure the threshold of the number of cores for which the MPI communications are prohibitive.

This work highlights the importance of finding a trade-off between different data structures and optimization techniques, in order to obtain overall gain.

In the future, we intend to adapt our vectorization techniques when dealing with other boundary conditions like reflecting or escaping particles. Next, we plan to port our algorithm to Many Integrated Core (MIC) architectures in order to use the 512 bits data registers for vectorization and to reinforce the multi-threading.

Finally, formulas also exist for space-filling curves in three dimensions. Thus, the efficient PIC code we developed in this work opens up the possibility to run simulations in several areas of plasma physics in a three-dimensional physical space.

Acknowledgment

The authors acknowledge M. Mehrenberger for the access to GENCI's supercomputer Curie through the DARI project. They also want to thank the anonymous referees whose interesting remarks improved the overall quality of the paper.

References

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

[2] K. J. Bowers and H. Li. "Helicity dissipation in 3d PIC simulation of magnetic reconnection of a force free configuration". 45th Annual Meeting of the APS Division of Plasma Physics. 2003.

[3] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation". In: *Physics of Plasmas* 15.5 (2008), p. 055703.

[4] E. Chacon-Golcher, S. A. Hirstoaga, and M. Lutz. "Optimization of Particle-In-Cell simulations for Vlasov-Poisson system with strong magnetic field". In: *ESAIM: Proceedings and Surveys* 53 (2016), pp. 177–190.

[5] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. "Nonlinear Array Layouts for Hierarchical Memory Systems". In: *Proceedings of the 13th International Conference on Supercomputing*. 1999, pp. 444–453.

[6] V. K. Decyk and T. V. Singh. "Particle-in-Cell algorithms for emerging computer architectures". In: *Computer Physics Communications* 185.3 (2014), pp. 708–719.

[7] V. K. Decyk, S. R. Karmesin, A. de Boer, and P. C. Liewer. "Optimization of particle-in-cell codes on reduced instruction set computer processors". In: *Computers in Physics* 10.3 (1996), pp. 290–298.

[8] D. DeFord and A. Kalyanaraman. "Empirical Analysis of Space–Filling Curves for Scientific Computing Applications". In: *42nd International Conference on Parallel Processing (ICPP)*. Oct. 2013, pp. 170–179.

[9] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. URL: http://www.fftw.org/.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms". In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. 1999, pp. 285–299.

[11] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, and A. Bhattacharjee. "The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing". In: *Journal of Computational Physics* 318 (2016), pp. 305–326.

[12] D. Hilbert. "Über die stetige abbildung einer linie auf ein flächenstück". In: *Mathematische Annalen* 38 (1891), pp. 459–460.

[13] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Institute of Physics, Philadelphia, 1988.

[14] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[15] J. Mellor-Crummey, D. Whalley, and K. Kennedy. "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings". In: *International Journal of Parallel Programming* 29.3 (2001), pp. 217–247.

[16] G. M. Morton. "A computer oriented geodetic data base and a new technique in file sequencing". In: *IBM Germany Scientific Symposium Series*. 1966.

[17] R. Raman and D. S. Wise. "Converting to and from Dilated Integers". In: *IEEE Transactions on Computers* 57.4 (2008), pp. 567–573.

[18] J. Skilling. "Programming the Hilbert curve". In: *AIP Conference Proceedings* 707 (2004), pp. 381–387.

[19] Innovative Computing Laboratory at the University of Tennessee. *Performance Application Programming Interface*. URL: http://icl.cs.utk.edu/papi.

[20] H. Vincenti, M. Lobet, R. Lehe, R. Sasanka, and J.-L. Vay. "An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes". In: *Computer Physics Communications* (2016), pp. 145–154.