Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors

Yann Barsamian[✉1,2] [0000−0001−6602−0547], Arthur Charguéraud[2,1] [0000−0001−7764−4507], Sever A. Hirstoaga[2,3], and Michel Mehrenberger[3,2] [0000−0001−6376−409X]

[1] Université de Strasbourg, CNRS, ICube UMR 7357 (Strasbourg, France)
[2] Inria (Nancy, France)
[3] Université de Strasbourg, CNRS, IRMA UMR 7501 (Strasbourg, France)
ybarsamian@unistra.fr, arthur.chargueraud@inria.fr, sever.hirstoaga@inria.fr, mehrenbe@math.unistra.fr

Abstract. Particle-in-Cell (PIC) codes are widely used for plasma simulations. On recent multi-core hardware, performance of these codes is often limited by memory bandwidth. We describe a multi-core PIC algorithm that achieves close-to-minimal number of memory transfers with the main memory, while at the same time exploiting SIMD instructions for numerical computations and exhibiting a high degree of OpenMP-level parallelism. Our algorithm keeps particles sorted by cell at every time step, and represents particles from a same cell using a linked list of fixed-capacity arrays, called chunks. Chunks support either sequential or atomic insertions, the latter being used to handle fast-moving particles. To validate our code, called Pic-Vert, we consider a 3d electrostatic Landau-damping simulation as well as a 2d3v transverse instability of magnetized electron holes. Performance results on a 24-core Intel Skylake hardware confirm the effectiveness of our algorithm, in particular its high throughput and its ability to cope with fast moving particles.

Keywords: Particle-in-cell · Plasma physics · Multi-core · SIMD architecture · Shared memory · Chunks · Strict binning · Magnetized electron holes

1 Introduction

The Particle-in-Cell (PIC) method enables large-scale simulations of plasma physics. PIC simulations are, for example, key to the design of ITER fusion reactor, and they also apply to other domains, e.g., astrophysics. As of 2018, PIC simulations accommodate at the order of 10^{13} particles, involving the hundreds of thousands of cores available on the world's top super-computers.

To reach such a scale, state-of-the-art PIC codes exploit parallelism available at three levels: inter-node parallelism using, e.g., MPI; shared-memory multi-threading using, e.g., OpenMP, and register-level parallelism using SIMD instructions. A number of implementations also leverage GPUs or MICs. Implementations include EMSES [17], GTC-P [20], ORB5 [12], OSIRIS [8], PICADOR [19], PIConGPU [5], PSC [10], VPIC [4].

A recent paper [20] studied GTC-P performances in details, and points out that: "*metrics such as flop/s or percentage-of-peak are less relevant for the predominantly memory-bound gyrokinetic PIC methods.*" The authors then present a model able to predict execution time as a function of data transfers. Most predominant is the intra-node operations on the shared memory (60% to 80% of the execution time). Their cost is decomposed between in-cache accesses, contiguous accesses, and random accesses—the latter being the most costly. This study shows that, to improve the performance of multi-core (intra-node) processing in PIC simulations, we must decrease the amount of costly memory accesses. Of course, we must do so by preserving the OpenMP-level parallelism as well as the crucial use of SIMD instructions.

The *strict-binning* approach to implementing the PIC method enables significant reduction in the number of random accesses and cache misses [21, 7, 17, 1]: at every time step, particles that fall in the same cell are stored together. Doing so brings two main benefits. First, the electric field values can be read only once per cell, avoiding numerous cache misses and allowing SIMD computations when updating particle velocities. Second, the representation saves the need to store a cell index for each particle, thereby saving memory loads and stores.

A central challenge with the strict-binning approach is the representation of the dynamically-sized bins storing the particles. A state-of-the-art proposal by Nakashima et al. [17] organizes particles in a big array, ordering them according to their cell index, and leaving variable-size *gaps* between the groups of particles associated with each cell. Yet, this approach suffers from two important limitations. First, as particles move, maintaining the variable-size gaps requires costly operations for shifting particles. Second, the algorithm, which uses a coloring scheme [13] to avoid data races when processing the cells in parallel, does not handle well *fast-moving particles* (particles moving more than a couple cells away at a given time step): it resorts to sequential processing for these particles.

These two limitations are exacerbated when the percentage of *crossing particles* (particles changing cells at each time step) increases, to the point of possibly becoming a major bottleneck. For example, in a parallel execution using 64 cores, having as little as 0.5% of fast-moving particles can result in a 32% slowdown on the total execution time due to the sequential processing of these particles.[1]

We propose an algorithm implementing strict-binning for the PIC method that addresses the aforementioned limitations, while still supporting efficient OpenMP/SIMD parallelization of all critical loops. Our algorithm leverages the use of *chunk bags*, i.e. linked lists of fixed-capacity arrays, to achieve SIMD-friendly storage of particles with limited memory overheads. These chunk bags are furthermore devised to support atomic push operations, which are used to handle fast-moving particles within the main parallel loop. Our algorithm minimizes the amount of memory transfers: at each time step, each particle gets read from and written to memory exactly once. In particular, no further move or reordering is ever required, regardless of the percentage of fast-moving particles.

[1] Let t denote the single-core execution time. Assume 0.5% of sequential execution, and 99.5% using 64 cores. The parallel execution time is: $0.005t + 0.995t/64 = 1.32t/64$.

Parameters

N: number of particles.
$\mathsf{nbCells} = X \cdot Y \cdot Z$: size of the grid.
Δt: duration of a time step.

Variables

$\mathsf{particles}[0..\mathsf{N}-1]$: set of particles, with position $\mathbf{x_p}$ and velocity $\mathbf{v_p}$.
$\rho[0..X][0..Y][0..Z]$: charge density.
$\mathbf{E}[0..X][0..Y][0..Z]$: electric field.

Algorithm

Foreach time step
  Set all cells of ρ to 0
  Foreach particle
    Interpolate $\mathbf{E}$ to $\mathbf{x_p}$ — Stored in $\mathbf{E_p}$
    Update velocity — $\mathbf{v_p}$ += $\frac{q}{m}\mathbf{E_p}\Delta t$
    Update position — $\mathbf{x_p}$ += $\mathbf{v_p}\Delta t$
    Accumulate charge from $\mathbf{x_p}$ on ρ
  Compute $\mathbf{E}$ from ρ — Poisson solver

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

This algorithm is efficient provided that the average number of particles per cell exceeds a couple hundreds. Although laser-driven particle acceleration simulations can use as few as 30 particles per cell [5], large-scale, high-precision simulations may involve hundreds to thousands of particles per cell [21, 4, 10].

Through the rest of this paper, we describe our algorithm, comment on its theoretical properties (space usage, parallelization of critical loops, amount of memory transfers), discuss performance results (bandwidth usage, impact of fast-moving particles), numerical results (simulation of Landau-damping and of transverse instability of magnetized electron holes), and related work.

2 An Efficient, Strict-Binning, Multicore PIC Algorithm

Fig. 1 describes the PIC method, applied to the resolution of the Vlasov-Poisson system, which models the time evolution of the distribution function of charged particles in a plasma. Following the Cloud-in-Cell model [3, Sect. 2.6.], we interpolate the electric field and accumulate the charges linearly from/to the eight corners of the grid cell where each particle lies. The Poisson solver takes less than 5% of the execution time, we thus focus our attention on the particle loop.

Our implementation performs all computations in double precision, with the exception of positions, which are stored using the "index plus offset" representation [4, III.E.], whereby the position of a particle is described relative to the corner of the cell containing the particle, using 3 `float` values, yielding sufficient precision. In the strict binning approach, the index of the containing cell is implicit. Thus, each particle admits a 36-byte representation.

Particles are stored in fixed-capacity arrays (*chunks*, e.g., [1]). Several chunks might be needed to store all the particles contained in a same cell. Each cell is thus described by a linked list of chunks (a *chunk bag*). The number of particles in a chunk, denoted by K thereafter, should enable efficient vectorization (K is a multiple of 16 for 512-bit registers), and at the same time be large enough to tame the cost of following a pointer from a chunk to the next (e.g., 128 or 256).

Benchmarking on the hardware considered reveals that the structure of arrays (SoA) layout, which enables better vectorization, improves performance compared with the array of structures (AoS) layout. The memory layout we use for the particles is summarized on the next page. (Arrays should be aligned.)

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

The chunk bag data structure supports $O(1)$ insertion of a particle (adding a fresh chunk if needed), $O(1)$ merge of two bags thanks to the *back* field (note that chunk compaction is not needed), and $O(n)$ iteration over the contents, all with excellent constant factors. Furthermore, unlike chunks introduced by prior work [1], our chunk bags are devised to support a thread-safe atomic insertion operation. Atomic insertions are central to the handling of fast-moving particles. Atomic insertion uses a fetch-and-add instruction to reserve a slot in the chunk for the particle. If a thread attempts to reserve the one-past-the-end slot, it acquires responsibility to extend the bag with a fresh chunk, in which case it sets the *next* pointer of the fresh chunk to the current *front* pointer of the bag, and sets the *front* pointer of the bag to the address of the fresh chunk.

When processing a chunk of particles, the algorithm first updates velocities and positions, then migrates the particles to different chunks, depending on the cell associated with their new position. Once all particles from the chunk are processed, the chunk is stored into a (per-core) free list, so as to be subsequently reused to extend a bag whose last chunk becomes full. Our algorithm preserves the following invariant: at the beginning of a time step, all the particles are stored in at most $\lceil \mathsf{N}/\mathsf{K} \rceil + 2 \cdot \mathsf{nbCells}$ chunks, where N denotes the total number of particles, and K denotes the number of particles per chunk.

To dispatch particles according to their target cells, we associate two bags with each cell: a *private bag*, accessed at most by one thread at a time; and a *shared bag*, accessed concurrently, to handle fast-moving particles. To initialize these two bags, we need an additional $2 \cdot \mathsf{nbCells}$ empty chunks. In total, we need $\lceil \mathsf{N}/\mathsf{K} \rceil + 4 \cdot \mathsf{nbCells}$ chunks. We have proved that this number of chunk suffices at any point of a simulation, regardless of how particles move. Thus, the space used by our algorithm, in addition to the minimal amount of memory needed to represent the particles, grows in proportion with $4\mathsf{K} \cdot \mathsf{nbCells} \cdot \mathsf{sizeof(particle)}$.[2]

In order to maximize the number of insertions into private bags while preserving a high degree of OpenMP parallelism, we follow the coloring scheme proposed by Kong et al. [13], and generalized from 2d to 3d by Nakashima et al. [17]. The idea is to fill the space with *tiles*, of size 2x2x2 (or more), in a regular manner. Tiles are colored using 8 different colors in such a way that two adjacent tiles have distinct colors. At each of the 8 color phases, $\frac{1}{8}$ of the tiles are processed, in parallel by nbCores threads.[3] Because cells processed in parallel by distinct threads are at least 2 cells away from each other, all the particles that

[2] In practice, we allocate a dozen extra chunks per core, giving some slack and avoiding dynamic load balancing of free chunks. Note that it is very unlikely for these chunks to ever be needed, because cores free chunks at a faster rate than they fill chunks.

[3] For a 2x2x2 tiling, at the i-th coloring phase, the algorithm processes cells whose coordinates satisfy: $((x/2) \bmod 2) + 2 \cdot ((y/2) \bmod 2) + 4 \cdot ((z/2) \bmod 2) = i$. Using larger tiles is possible but greatly reduces the number of tiles processed in parallel.

```
1  bag particles[0..nbCells−1]; // Particles by cell, at current time step
2  bag particlesNextPrivate[0..nbCells−1], particlesNextShared[0..nbCells−1];
3  double ρ[0..X][0..Y][0..Z], E[0..X][0..Y][0..Z];
4  double ρNext[0..nbCores−1][0..nbCells−1][0..7]; // 8 corners per cell
5  Foreach time step
6    Foreach color in [0..7] // 8 coloring phases
7      Parallel Foreach tile of that color // OpenMP parallel
8        Foreach cell idCell in that tile
9          Read E[x][y][z], foreach (x, y, z) among the 8 corners of cell idCell
10         Foreach chunk in particles[idCell]
11           Foreach particle in that chunk // SIMD vectorized
12             Update particle velocity
13           Foreach particle in that chunk // SIMD vectorized
14             Update particle position
15             Compute idCellNext, the index of the cell containing the particle
16           Foreach particle in that chunk
17             If the particle moves inside its tile
18             Or it moves to the closer half of a neighbor tile
19               Add the particle into particlesNextPrivate[idCellNext]
20             Else
21               Atomically add the particle into particlesNextShared[idCellNext]
22             Add its charge into ρNext[thisCoreId][idCellNext][..] // SIMD
23         Put a pointer to that chunk into the freelist of the current core
24   Parallel Foreach idCell in [0..nbCells−1] // OpenMP parallel
25     Set particles[idCell] to particlesNextPrivate[idCell]
26     Merge particlesNextShared[idCell] into particles[idCell]
27     Set particlesNextPrivate[idCell] to empty, using an empty chunk
28     Set particlesNextShared[idCell] to empty, using an empty chunk
29   Parallel Foreach (x, y, z) in [0..X]x[0..Y]x[0..Z] // OpenMP parallel, collapsed
30     Foreach of the 8 pairs (idCell,i) such that (x,y,z) is i−th corner of idCell
31       Foreach idCore in [0..nbCores−1]
32         ρ[x][y][z] += ρNext[idCore][idCell][i]
33         ρNext[idCore][idCell][i] = 0
34   Compute E from ρ using a Poisson solver and set ρ to 0 // FFTW + OpenMP
```

Fig. 2. Our parallel algorithm for the PIC method on multicore architectures.

move, at a given time step, no more than one cell away (no more than half a tile away, in general) can be pushed into private bags, in a thread-safe manner.

The pseudo-code of our algorithm appears in Fig. 2. Particles from a same cell are processed sequentially by a same thread. To benefit from SIMD performance, we split the loop over each chunk. (If one chunk does not fit into L1 cache, additional splitting is needed.) First, our code updates velocities (line 12). Second, it computes the new positions (line 14), introducing an auxiliary array for storing the new cell indices. Third, it sequentially pushes each particle into the chunk associated with its target cell. If the target cell lies in the current tile, or lies in the closer half of an immediate neighboring tile, a non-atomic insertion is performed on a private bag (line 19). Otherwise, an atomic insertion is

performed on a shared bag (line 21). Note that the boolean condition involved can be evaluated using a simple arithmetic test.

Once all the particles are processed, the algorithm merges, for each cell, its private bag with its shared bag (line 26). No chunk compaction is performed at this point; as a result, the bag associated with one cell may contain up to 2 non-full chunks. Thus, there are at most $\lceil \mathsf{N}/\mathsf{K} \rceil + 2 \cdot \mathsf{nbCells}$ nonempty chunks at the beginning of the next time step. It follows that at least $2 \cdot \mathsf{nbCells}$ empty chunks must have been freed during the current time step. This number corresponds exactly to the number of chunks needed to initialize the private and the shared bags for the next time step. Our algorithm performs this initialization efficiently in parallel (using a prefix sum array, based on the sizes of the per-core free lists).

We next describe the treatment of the charge density and the electric field (ρ and E). When processing particles from one cell, the algorithm first reads from memory the values of the electric field on the 8 corners of that cell (line 9). Importantly, thanks to the strict-binning approach, this data needs only be loaded once from memory. As particles are processed and moved to their target cells, the charge of each particle is accumulated (line 22) into the array $\rho\mathsf{Next}$, which, at the end of the time step, is used to update E for the next iteration. We exploit a recently-proposed, ingenious technique allowing to accumulate the charge on the 8 corners using SIMD instructions [22]. Concretely, the array $\rho\mathsf{Next}$ involves some amount of redundancy: for each cell, 8 values are stored adjacently in memory, describing the charge on the 8 corners of that cell. At the end of a time step, the charge at a grid point is computed by summing the values associated with the 8 cells that have this grid point as one of their corners (line 32).

We considered two different possibilities for updating $\rho\mathsf{Next}$. The first possibility is to decompose $\rho\mathsf{Next}$ into a *private* array and a *shared* array, just like we do for bags of particles. In this approach, only the deposit of the charge of fast-moving particles triggers atomic operations; for all others particles, we can use SIMD operations. The second possibility is to decompose $\rho\mathsf{Next}$ into $\mathsf{nbCores}$ arrays. In this approach, each core has exclusive access to its charge array, so all accesses use SIMD operations. The downside is a slight increase in the memory footprint, and in the time needed to sum up the values. However, under our assumption of a reasonably large number of particles per cell, these additional costs are tiny in front of the gains. Thus, we opted for the latter approach.

Under the assumption of (at least) hundreds of particles per cell in average, the operations for manipulating chunks (following pointers, pushing/popping in free lists) and for manipulating per-cell information are all well amortized. Overall, our algorithm is not far from optimal in terms of memory transfers.

Optimization when particles move at most one cell per time step. For simulations whose physical parameters ensure that movement is restricted to immediate neighboring cells (e.g., [4, 10]), we can optimize our algorithm by removing the shared bags altogether. In this case, we require only $\lceil \mathsf{N}/\mathsf{K} \rceil + 2 \cdot \mathsf{nbCells} + \mathsf{nbCores}$ chunks, and do not need any atomic insertion operation. Likewise, $\rho\mathsf{Next}$ can be stored in a single array (indexed by cells and by corners).

3 Performance Results

To assess correctness and performance of our code, which is called Pic-Vert, we considered two classical test cases: a 3d Landau-damping simulation and a 2d3v electron hole simulation. Section 4 presents details on these experiments, and argues that the numerical results produced by our simulation match the expected results. In the remaining of this section, we discuss performance results.

Experimental hardware 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, and the FFTW3 library [9] for the Poisson solver.

The algorithm depends on two parameters. First, we use tiles of size `2x2x2` for the coloring. Tiles of size `4x4x4` lead to similarly good performances. Using larger tiles degrades performance. Second, we use $\mathsf{K} = 256$ for the chunk capacity. Larger values of K increase the space usage and do not reduce the execution time. Smaller values of K increase the execution time overheads: +12% for $\mathsf{K} = 128$, and +52% for $\mathsf{K} = 64$. Note that, for $\mathsf{K} = 256$, the memory "slack", which is equal to $4 \cdot \mathsf{nbCells} \cdot \mathsf{sizeof(chunk)}$, represents in the Landau-damping simulation only 13% of the amount of memory strictly required for representing the particles.

Achieved throughput. For the end-user of a simulation, the metric that matters is the number of particles processed per second. The Pic-Vert code achieves:

- 740 million particles per second (30.8m/s/core) in the 3d Landau-damping simulation, where 31% of the particles change cell at each iteration;
- 910 million particles per second (37.9m/s/core) in the 2d3v electron hole simulation, where 32% of the particles change cell at each iteration.

Analysis in the roofline performance model. As argued in Sect. 2, our algorithm performs not far from the minimal number of memory operations—a key feature for PIC simulations hit by the memory bandwidth bottleneck. With this property in mind, it is interesting to compare the memory bandwidth achieved by our algorithm against the capacity of the hardware. Consider the Landau-damping simulation. The memory bandwidth achieved is 53.6 GB/s.[4] The *theoretical peak* advertised by the manufacturer is 127.99 GB/s. The Stream benchmark [15], which aims at evaluating the *practical peak* using a few microbenchmark programs, and which is commonly used as a baseline, provides the measure 98.2 GB/s. Our algorithm thus achieves 42% of the theoretical peak and 55% of the practical peak bandwidth. Reaching higher percentage in a PIC simulation appears to be very challenging.

Our algorithm is memory bound. In general, an algorithm may be *compute bound* (i.e. limited by the number of floating-point operations per second) or

[4] The bandwidth is obtained by multiplying the size of a particle (36 bytes, plus $\frac{64}{\mathsf{K}}$ bytes to account for chunk headers) by the number of particle processed per second (740 million), and by a factor 2 (one read plus one write). It would be very interesting to compare with other algorithms. Unfortunately, such numbers are rarely advertised, and comparisons are often hazardous due to differences in hardware, in particle representation (e.g., `float` vs `double`), in numerical schemes, etc.

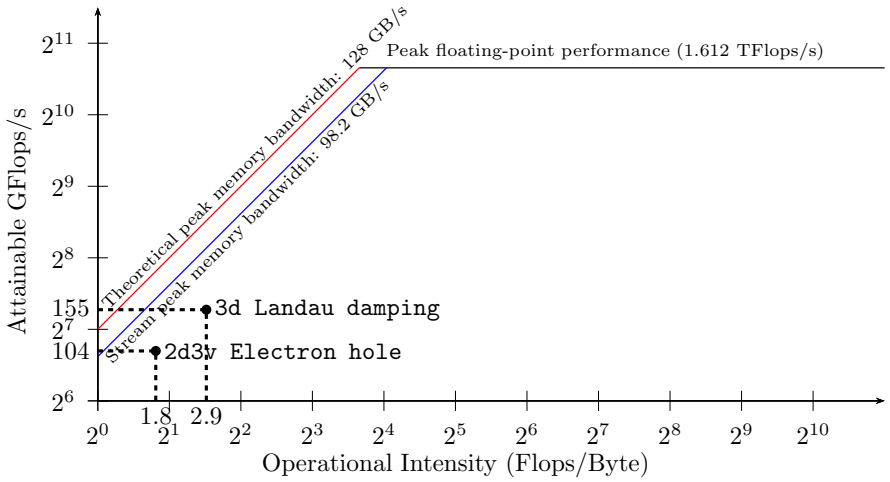

Fig. 3. Analysis of performances in the roofline model.

memory bound (i.e. limited by the number of bytes per second transfered from main memory) depending on its *operational intensity*, defined as the number of operations performed divided by the number of bytes moved from or to the main memory. We computed the operational intensity of the 3d code by counting the number of floating point operations per particle (79 operations in single-precision and 65 in double-precision, which leads to 209 operations when normalized to single-precision), and counting the number of bytes used to represent a particle (36 bytes, plus 0.25 byte to account for chunk headers). We thus derive that our 3d code has an operational intensity equal to $209/(2 \cdot 36.25) \approx 2.9$. Similarly, we computed the operation intensity for the 2d3v code to be $114/(2 \cdot 32.25) \approx 1.8$.

Fig. 3 represents the bounds on computation and memory bandwidth, in a chart showing the operational intensity on the x-axis, and the computation performance on the y-axis [23]. Note that both axes are log-scale. The computation bound is an horizontal line, at 1,612 GFlops/s (billion floating-point operations per second), a figure provided by the hardware manufacturer. The theoretical and practical memory bounds (bytes/s) are diagonal lines, because the bound in performance (flop/s) is equal to the operational intensity (flop/byte) multiplied by the memory bandwidth (bytes/s). Each diagonal line meets the horizontal line at the point of break-even between memory bound and compute bound.

Efficient processing of fast-moving particles. In addition to being memory efficient, our algorithm also benefits from another key feature not found in prior strict-binning algorithms: fast-moving particles are handled efficiently within the main parallel loop. For a particle moving more than half a tile away, we require only one extra atomic operation. Moreover, the contention associated with this atomic operation is relatively limited. Indeed, for two atomic operations to be issued on the same memory cell at the "same time" (i.e., close enough in time for a race on the cache line to occur), it must be the case that two particles taken

Particles that move 1 cell away	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%
Particles that move 2 cells away	0	0.7%	1.9%	3.1%	4.3%	5.6%	4.4%
Particles that move 3 cells away	0	0	0	0	0	0.2%	1.4%
Particles pushed atomically (line 21)	0.0%	0.4%	1.0%	1.6%	2.2%	3.1%	3.7%
Slowdown w.r.t. first column	0	0.0%	0.9%	3.8%	4.4%	4.2%	7.0%

Table 1. Impact on performance of increasing the percentage of fast particles.

from two distinct tiles spaced away by at least one full tile are moving towards the same cell of a third tile, at the "same time". Thus, the performance of our algorithm should be relatively independent form the heat.

To empirically evaluate the impact of fast-moving particles, we consider a simulation in which we artificially varied the initial distribution of particle velocities. To that end, we manually tuned these distributions in such a way as to obtain several test cases with increasing number of fast-moving particles.[5] Each test case is reflected by a column from Table 1. More specifically, the three first rows show the percentage of particles that move away from 1, 2 or 3 cells from their current grid cell at each time step (no particle move further away).

By instrumenting the code, we measured the number of push operations that trigger an atomic write (line 21 from Fig. 2). These numbers, relative to the total number of particles, appear in the fourth line of the table: they vary from 0% to 3.7%. The last row of Table 1 gives the corresponding slowdown on the total execution time. Figures show that even when the percentage of particles whose move require an atomic operation is as high as 3.7%, the cost of processing these fast moving particles remains fairly limited: +7.0%. In comparison, any alternative algorithm that sequentially processes 3.7% of the particles in a 24-core execution would suffer at least from a +85% slowdown (recall Sect. 1).

Scaling. Although inter-node parallelism is orthogonal to the focus of the present paper, we used particle decomposition to scale our algorithm on 128 Skylake sockets (each with 24 cores, 12.3 TB of RAM in total), using one MPI process per socket. We simulated Landau-damping with 256 billion particles, achieving a throughput of 89.6 billion particles per second: 123x speedup w.r.t. one socket.

4 Numerical Results

3d3v Landau-damping. We consider a classical Landau-damping test case [3, 11], simulating 2 billion particles on a 64×64×64 grid, for 500 time steps. We use the same parameters as in [18]: time step of 0.05, periodic boundary conditions on spatial domain $\Omega = [0, 22]^3$ and initial distribution function:

$$f_0(x, y, z, \mathbf{v}) = \frac{1}{(2\pi)^{3/2}} e^{-|\mathbf{v}|^2/2} L(x)L(y)L(z) \quad \text{with } L(w) = 1 + 0.05\cos(\pi w/11).$$

Fig. 4 represents the evolution of electric energy. It shows that the decay slope in our simulation is in accordance with the theoretical value $\gamma = -0.008466$ obtained from the dispersion analysis.

[5] Particle velocities in the experiment of Table 1 follow the sum of two Gaussian distributions, like in the bump-on-tail instability. Details may be found in [2].

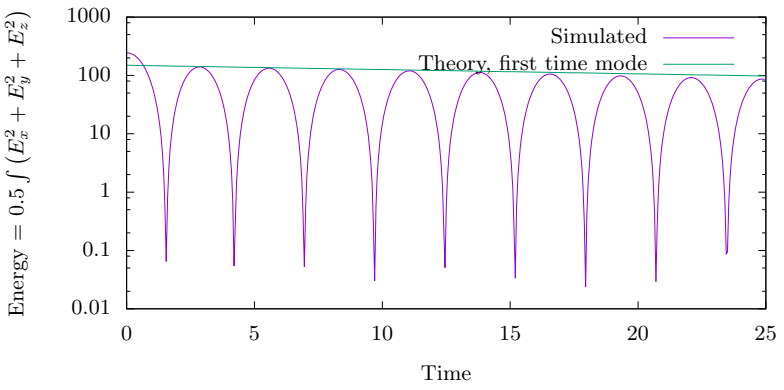

Fig. 4. Time evolution of electric energy in the Landau damping simulation.

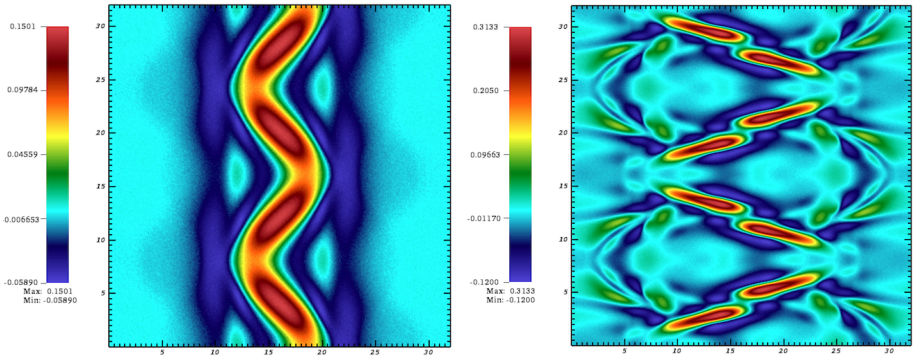

Fig. 5. Time evolution of $\rho(t, x, y)$ in the electron hole simulation, at $t = 20$ and $t = 40$.

2d3v Electron hole. We consider a more complex test case proposed by Muschietti et al. [16]. We simulate 64 billion particles on a 512×512 grid (on 32 Skylake sockets). Time step is 0.1 and spatial domain is $[0, L]^2$, with $L = 32$. The initial function is $f(x, y, v) = F_1(v_1^2 - 2\phi(x, y))e^{-50(v_2^2+v_3^2)}$ with potential $\phi(x, y) = e^{-0.5\left((x-L/2)/\Delta_{\|} - 0.3\cos(0.39y)\right)^2}$, with $\Delta_{\|} = 3$ and F_1 defined as:

$$F_1(w) = \begin{cases} \frac{\sqrt{-w}}{\pi\Delta_{\|}^2}\left(1 + 2\ln(\frac{\psi}{-2w})\right) + \frac{6+(\sqrt{2}+\sqrt{-w})(1-w)\sqrt{-w}}{\pi(\sqrt{2}+\sqrt{-w})(4-2w+w^2)}, & \text{for } -2\psi \le w < 0, \\ \frac{6\sqrt{2}}{\pi(8+w^3)}, & \text{for } w > 0. \end{cases}$$

The external magnetic field is aligned with x and has amplitude $B_0 = 0.2$.

Fig. 5 shows the charge density $\rho(t, x, y) = 1 - \int f(t, x, y, v)dv$, on the left at time $t = 20$, and on the right at time $t = 40$. These results are qualitatively similar to those from Muschietti et al. [16].

In addition, we studied the convergence of the simulation with respect to the number of particles and to the grid size. To that end, we compare, for different settings of these two parameters, the time evolution of a quantity representative

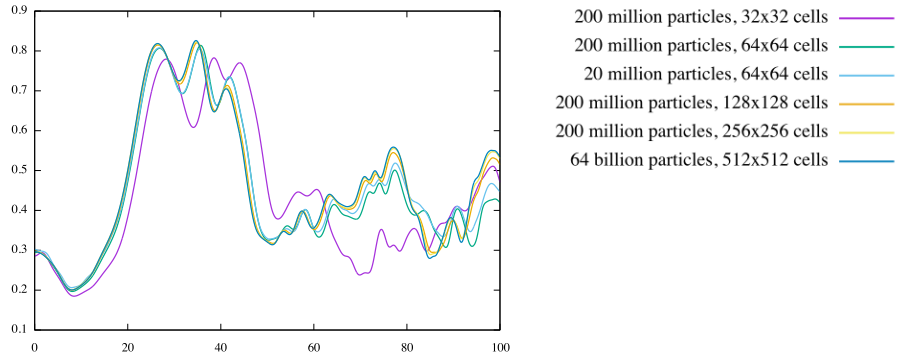

Fig. 6. Time evolution of the y part of electric field norm in the electron hole simulation, for different values of the number of particles and of the grid size.

of the instability.[6] Results appear in Fig. 6. They show that using a small 32×32 grid with 200 million particles as considered by Muschietti et al. exhibits the correct qualitative behavior up to $t = 50$, but diverges beyond this point.

For a quick simulation, it appears preferable to use a 64×64 grid with only 20 million particles, as it gives quantitatively accurate results up to $t = 50$. For longer simulations, our results show that using a 128×128 or a 256×256 grid with 200 million particles suffices to give accurate results up to $t = 100$. Indeed, the two corresponding curves are close to that of our large-scale simulation, which uses a 512×512 grid with 64 billion particles (the top-most curve at $t = 100$).

5 Technical Comparison to Related Work

We organize the discussion of related work by focusing on three main criteria: strict or non-strict binning, representation of particles, and treatment of data races arising when two threads push data onto a same target cell.

To ensure efficient accesses to the electric field and charge arrays, locality is essential in PIC simulations. In numerous algorithms, particles are stored in an array and ordered by their cell index. Yet, because particles move in the grid, their locality in the array decreases at each time step. Thus, reordering operations must be performed: either at every time step, to maximize locality (e.g., [12]), or at some lower frequency, reducing locality but mitigating the cost of re-sorting (e.g., [4, 10]). Depending on the option, performances suffer either from numerous costly random accesses or from suboptimal locality.

Rather than sorting, other algorithms rely on *coarse-grained binning* [5, 20, 19, 22]. Particles are organized in super-cells (of size, e.g., `10x10x10`), and a dynamically-sized data structure is used to represent particles from a same super-cell. For example, *attribute tiles* [5] have been used to store particles on GPU

[6] This quantity, which we call "y part of electric field norm", is defined as half of the square root of the electric energy $\int (E_x^2 + E_y^2)$ minus the part of that energy corresponding to the modes in x (here, the first 20 modes).

using doubly linked lists of fixed-capacity arrays. Unlike with our chunks, particles in an attribute tile are processed in place. If a particle moves to a different super-cell, it is migrated to a transfer buffer, and its slot is marked as a *hole* in an auxiliary bitmap. Subsequently, holes are filled with particles incoming from neighboring super-cells. Remaining holes, if any, are filled using particles taken from the end of the attribute tile. In contrast, in our algorithm, particles are directly moved to their target bin—they get moved exactly once per time step.

While *strict binning* overcomes several of the aforementioned limitations, it is nontrivial to implement efficiently. Representing each cell by a single fixed-capacity array [21, 6] is space inefficient, and falls short outside of specific scenarios where uniform particle density can be assumed. Linked lists [11, Sect. 8.4.] lead to tremendous overheads both in terms of space (to represent list cells) and time (to follow indirections). Vectors, a.k.a. resizable arrays, suffer from a prohibitive 2x space overhead and involve costly resize operations.

Packed Memory Arrays (PMAs) [7] have been proposed as a specialized data structure for keeping particles sorted by their cell index. This data structure stores particles in a large array that also contains a fraction of unused cells, called *gaps* (a.k.a. *holes*). The width of the gaps may increase or decrease as particles move cells. When a gap closes, rebalancing operations involving backward or forward shifting of the particles must be performed to restore balanced gaps.

Nakashima et al. [17] propose a data structure that we view as a parallelism-friendly version of PMAs. To tame the frequency of rebalancing operations, the authors introduce thread-local *overflow buffers*. However, as the authors acknowledge [17, III.D.], these buffers come at the cost of increased complexity in the code, of additional space usage, and of slower processing of the overflow buffers, on which SIMD operations do not apply.

In Nakashima et al.'s work [17], most data races are eliminated thanks to the use of a 8-color scheme [13], which we also use. For the remaining data races, which are associated with fast-moving particles, their algorithm processes them in a separate sequential loop, which induces a major sequential bottleneck as soon as the percentage of fast-moving particles exceeds a fraction of a percent. In contrast, we are able to integrate the processing of these particles within the main parallel loop, using an atomic operation.

Furthermore, unlike prior work exploiting PMAs, our approach relies on a general-purpose bag representation, based on chunks. We only customize the bag implementation to accommodate SoA layout. Our data structure does not involve any shifting of data nor any overflow buffer. This has two main benefits. First, we save numerous memory operations. Second, the performance of our algorithm is robust to an increase in the percentage of fast-moving particles.

6 Future Work

In this work, we focused on multicore and SIMD parallelism. In future work, it would be great to extend our algorithm with a layer of domain decomposition, using MPI communications. We speculate that chunks could be used as buffers for emission and reception of particles reaching the cells at the frontier of a

domain. These chunks could then be merged, at the end of the time step, with the locally-processed chunks. The flexibility offered by chunks might be helpful for dealing with dynamically-sized domains. Furthermore, it would be interesting to adapt our algorithm to target architectures with larger number of cores, such as GPUs or MICs. We think that the organization in chunks could help addressing the issue of load balancing, which is critical on these architectures (e.g., [14]).

Data Availability Statement. The datasets and code generated during and/or analyzed during the current study are available in the figshare repository [2].

Acknowledgments. We would like to thank the anonymous reviewers for their valuable suggestions and comments. This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom Research and Training Programme 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, A. Charguéraud, and A. Ketterlin. "A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell Method". In: *Parallel Processing and Applied Mathematics: 12th Intl. Conf. (PPAM)*. 2018, pp. 133–144. DOI: 10.1007/978-3-319-78024-5_13.

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

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

[4] 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. DOI: 10.1063/1.2840133.

[5] 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, and R. Widera. "Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability". In: *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. 2013, 5:1–5:12. DOI: 10.1145/2503210.2504564.

[6] V. K. Decyk and T. V. Singh. "Particle-in-Cell algorithms for emerging computer architectures". In: *Comput. Phys. Commun.* 185.3 (2014), pp. 708–719. DOI: 10.1016/j.cpc.2013.10.013.

[7] M. Durand, B. Raffin, and F. Faure. "A Packed Memory Array to Keep Moving Particles Sorted". In: *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*. 2012. DOI: 10.2312/PE/vriphys/vriphys12/069-077.

[8] R. A. Fonseca, J. Vieira, F. Fiuza, A. Davidson, F. S. Tsung, W. B. Mori, and L. O. Silva. "Exploiting multi-scale parallelism for large scale numerical modelling of laser wakefield accelerators". In: *Plasma Phys. Control. Fusion* 55.12 (2013), p. 124011. DOI: 10.1088/0741-3335/55/12/124011.

[9] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. DOI: 10.1109/JPROC.2004.840301. URL: http://www.fftw.org.

[10] 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: *J. Comput. Phys.* 318 (2016), pp. 305–326. DOI: 10.1016/j.jcp.2016.05.013.

[11] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Institute of Physics, Philadelphia, 1988. DOI: 10.1201/9781439822050.

[12] A. Jocksch, F. Hariri, T.-M. Tran, S. Brunner, C. Gheller, and 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.

[13] X. Kong, M. C. Huang, C. Ren, and V. K. Decyk. "Particle-in-cell simulations with charge-conserving current deposition on graphic processing units". In: *J. Comput. Phys.* 230.4 (2011), pp. 1676–1685. DOI: 10.1016/j.jcp.2010.11.032.

[14] A. Larin, S. Bastrakov, A. Bashinov, E. Efimenko, I. Surmin, A. Gonoskov, and I. Meyerov. "Load Balancing for Particle-in-Cell Plasma Simulation on Multicore Systems". In: *Parallel Processing and Applied Mathematics: 12th Intl. Conf. (PPAM)*. 2018, pp. 145–155. DOI: 10.1007/978-3-319-78024-5_14.

[15] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Tech. Cmte. on Computer Arch. Newsletter (TCCA)* (1995), pp. 19–25. URL: https://www.cs.virginia.edu/stream/.

[16] L. Muschietti, I. Roth, C. W. Carlson, and R. E. Ergun. "Transverse Instability of Magnetized Electron Holes". In: *Phys. Rev. Lett.* 85.1 (2000), pp. 94–97. DOI: 10.1103/PhysRevLett.85.94.

[17] H. Nakashima, Y. Summura, K. Kikura, and Y. Miyake. "Large Scale Manycore-Aware PIC Simulation with Efficient Particle Binning". In: *2017 IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 2017, pp. 202–212. DOI: 10.1109/IPDPS.2017.65.

[18] L. F. Ricketson and A. J. Cerfon. "Sparse grid techniques for particle-in-cell schemes". In: *Plasma Phys. Control. Fusion* 59.2 (2017), p. 024002. DOI: 10.1088/1361-6587/59/2/024002.

[19] I. Surmin, S. Bastrakov, Z. Matveev, E. Efimenko, A. Gonoskov, and I. Meyerov. "Co-design of a Particle-in-Cell Plasma Simulation Code for Intel Xeon Phi: A First Look at Knights Landing". In: *16th Intl. Conf. on Algorithms and Architectures for Parallel Processing Workshops (ICA3PP, SCDT)*. 2016, pp. 319–329. DOI: 10.1007/978-3-319-49956-7_25.

[20] W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez, and T. Williams. "Extreme Scale Plasma Turbulence Simulations on Top Supercomputers Worldwide". In: *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 2016, pp. 502–513. DOI: 10.1109/SC.2016.42.

[21] D. Tskhakaya and R. Schneider. "Optimization of PIC codes by improved memory management". In: *J. Comput. Phys.* 225.1 (2007), pp. 829–839. DOI: 10.1016/j.jcp.2007.01.002.

[22] 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: *Comput. Phys. Commun.* 210 (2016), pp. 145–154. DOI: 10.1016/j.cpc.2016.08.023.

[23] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76. DOI: 10.1145/1498765.1498785.