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

Yann Barsamian✉[1,2], Arthur Charguéraud[2,1], and Alain Ketterlin[1,2]

[1] Université de Strasbourg, CNRS, ICube UMR 7357 (Strasbourg, France)
[2] Inria (Nancy, France)
ybarsamian@unistra.fr, arthur.chargueraud@inria.fr, alain@unistra.fr

Abstract. The Particle-in-Cell (PIC) method allows solving partial differential equation through simulations, with important applications in plasma physics. To simulate thousands of billions of particles on clusters of multicore machines, prior work has proposed hybrid algorithms that combine domain decomposition and particle decomposition with carefully optimized algorithms for handling particles processed on each multicore socket. Regarding the multicore processing, existing algorithms either suffer from suboptimal execution time, due to sorting operations or use of atomic instructions, or suffer from suboptimal space usage. In this paper, we propose a novel parallel algorithm for two-dimensional PIC simulations on multicore hardware that features asymptotically-optimal memory consumption, and does not perform unnecessary accesses to the main memory. In practice, our algorithm reaches 65% of the maximum bandwidth, and shows excellent scalability on the classical Landau damping and two-stream instability test cases.

Keywords: Particle-in-Cell simulation; plasma physics; strong scaling; weak scaling; hybrid parallelism; SIMD architecture.

1 Introduction

The Particle-in-Cell (PIC) method allows for simulations of a wide range of phenomena in plasma physics. For instance, it may be used to simulate the motion of a set of charged particles. In a PIC simulation, time is discretized, and the electric field is approximated using a grid. At each time step, each particle is accelerated with respect to that electric field, and moves according to its velocity. At its new location, each particle contributes its charge to the electric field, locally approximated using a grid. The resulting electric field is then involved at the next time step for accelerating particles [6, 16].

To increase the accuracy of a simulation, it is desirable to simulate as many particles as can be fit in the memory, and to perform as many time steps as possible. Thus, typically, both the memory and the execution time are limiting factors for such simulations. Practical simulations involve billions of particles (technically, of super-particles that approximate a set of nearby particles), involve large grids with millions of cells, and execute for thousands of time steps.

The challenge is to leverage the computing power of clusters of modern multicore hardware, where parallelism is available at two levels: across several machines, and among the cores of a same processor. The two levels differ significantly in that cross-machine communication is by several orders of magnitude more costly than in-machine communication through the shared memory. This difference explains the success of hybrid algorithms, which adopt two or three different strategies for efficiently exploiting all the parallelism available.

When the grid over which the simulation takes place is very large, the cost of maintaining a copy of the entire grid on every machine is prohibitive. In such situations, one resorts to domain decomposition, thereby assigning the available machines to subdomains of the grid space. A first challenge in domain decomposition is to balance the load. Possible solutions involve space-filling curves [1, 14], Barnes-Hut trees (or Octrees) [2, 26], or rectilinear partitioning (i.e., using parallelepipeds) [21, 22]. A second challenge is associated with the significant amount of communication involved for redistributing the particles that move across the subdomain boundaries. A typical plasma simulation may involve a significant fraction of fast-moving particles that frequently cross subdomains boundaries, thus requiring heavy cross-machine communication.

When the grid is not too large, particle decomposition may be used: particles are distributed evenly to the machines, each of which replicates the description of the electric field. The machines synchronize at every time step, by communicating the contribution of their particles to the charge density, in order to update the electric field. A successful hybrid approach, known as domain cloning [18, 23], consists of using domain decomposition in order to create subdomains that are just small enough for particle decomposition to apply.

Assuming the use of domain cloning, there remains need for an efficient algorithm to process the set of particles hosted on a single multicore machine with shared memory. Designing an efficient multicore algorithm for this core processing is the focus of this paper. Prior work on multicore algorithms for the PIC method have argued that storing particles in memory according to their position in the grid may yield significant benefits with respect to the locality of memory accesses, despite the cost of sorting. Moreover, prior work observes that if the particles are, at all times, grouped according to the cell in which they lie, then one may save the need to store, for each particle, the index of its containing cell.

For the benefits of locality to outweigh the cost of sorting, the sorting algorithm needs to be parallel and carefully optimized. Prior work (detailed in Sect. 2) has investigated the use of parallel versions of radix sort and counting sort, and optimizations of these to take into account the fact that not all particles move to remote cells. Prior work has also attempted by various means to reduce the memory usage associated to PIC codes, and to limit the number of synchronization barriers and of atomic instructions (e.g., compare-and-swap and fetch-and-add), which are more costly than conventional read-write operations.

All the PIC parallel algorithms targeting multicore architectures, as far as we know, suffer from at least one of two problems. (1) Many algorithms are suboptimal in the execution time. On the one hand, algorithms that do not reorder particles during the simulation suffer from poor locality, which leads to a higher number of cache misses and limits opportunities for vectorized processing. On the other hand, algorithms that do reorder particles—using sorting or maintaining buckets of nearby particles—involve nontrivial operations. In particular, the use of buckets is challenging in the context of a parallel algorithm: if buckets are per-thread, they need to be merged eventually; and if buckets are shared, they require expensive atomic operations for synchronization. (2) Many algorithms are suboptimal in space usage. On the one hand, algorithms that do not maintain particles sorted by cell index at all times require extra space to store those indexes for each particle. On the other hand, algorithms that do reorder particles typically involve auxiliary arrays to perform out-of-place sorting, or involve arrays with spare capacity to deal with the variability of the cardinality of each bucket. Since both the execution speed and the memory consumption are limiting factors in PIC simulations, we believe that there is space for improvement.

This work presents a novel parallel algorithm (detailed in Sect. 3) for PIC simulations on multicore architectures, featuring at the same time: asymptotically-optimal memory consumption, minimal bandwidth usage, competitive constant factors on the execution time, and excellent scalability. More precisely:

- Our algorithm reads and writes each particle exactly once from the main memory at each time step, thus is optimal in terms of memory transfer.
- Our algorithm requires, in addition to the minimal amount of space required for storing the particles, a space overhead that is constant for a fixed grid and a fixed hardware. In particular, our space usage does not depend on the number of particles that cross cell boundaries.
- Our algorithm allows all the cells of the grid to be treated in parallel, exposing an amount of parallel threads sufficient to feed all the cores, even in the face of relatively non-uniform distribution of the particles in space.
- Our algorithm involves only 3 synchronization points per time step, and it does not require any atomic operation.

The experiments (detailed in Sect. 4) performed on a 18 core, 2.3 GHz machine, on a 128x128 grid, show the following results:

- Compared with a carefully optimized, vectorized implementation of the standard approach that consists of assigning particles to cores, and sorting the particles every 20 time steps (frequency found to be optimal) our algorithm is 13% slower on a single core execution, but 36% faster on a 18 core execution. We explain the better scalability by the fact that we perform fewer memory accesses, and thus put much less pressure on the memory bus.
- Our algorithm reaches 65% of the maximal achievable bandwidth, as measured by the Stream reference benchmark. Given that our algorithm performs as few accesses as possible to the main memory, we conclude that there remains limited space for further improvements of the execution time.
- In terms of strong scaling, for a given input of 900 million particles, our algorithm achieves a 14.6x speedup on 18 cores, relative to its execution on a single core. In terms of weak scaling, our algorithm is only 18% slower

for simulating 1,800 million particles with 18 cores than it is for simulating 100 million particles with a single core.

- In terms of raw performance, our algorithm, when executed on 18 cores, processes 861 million particles per second. Equivalently, one core is able to process one particle at one time step in no more than 48 cycles, all inclusive.

In addition to our experiments on a single machine, we studied scalability on up to 128 sockets (64 dual-socket machines), each socket hosting 18 cores. We followed the particle decomposition approach, with each socket storing a copy of the electric field grid. As soon as all sockets have updated their charge density, we rely on a global MPI reduction to allow each socket to obtain the sum of the charge densities of all the sockets. Each socket then uses this total charge density to update the electric field. An execution involving 128 sockets and 128 times more particles is only 8% slower than an execution on a single socket, thus demonstrating excellent scalability. Overall, using the 2,304 cores available on the 64 machines, we are able to successfully simulate 230 billion particles ($2.3 \cdot 10^{11}$) for 100 iterations in no more than 228 seconds.

One key ingredient in our approach is the use of an optimized *bag* data structure for storing particles. A bag is essentially a linked list of fixed-size arrays, called chunks. Practice shows that chunks with a capacity of 512 particles yield optimal results. Our bags are thus extensible containers, with a fixed memory overhead—at most the size of an empty chunk. These bags support efficient iteration, essentially as fast as with a static array. Most importantly, chunks may be freed while iterating over the elements of the bag. This possibility enables us to perform our operations as in an out-of-place algorithms, yet without having to pay for the twofold space overhead associated with out-of-place algorithms.

At a given iteration of the simulation, we use one bag per cell from the grid, for storing the particles in this cell. To prepare for the next iteration, we need to distribute particles to different bags, which are associated with the next iteration. In order to avoid data races between the several cores that move the numerous particles, we allocate one bag for each cell and for each core. Once all particles are distributed in these bags, we merge, for each cell, the bags associated with that cell (there are as many such bags as cores). Since each merge operation takes constant time, as it amounts to an in-place concatenation of two linked lists, the overall cost of merging all these bags is $\mathcal{O}(\mathsf{nbCores} \times \mathsf{nbCells})$. This cost is, in practice, small compared to the processing of all the particles. Once the bags are merged, the particles are readily sorted for the next iteration.

One might worry about the memory overhead associated with the numerous bags involved. Yet, the total memory footprint of our algorithm is equal to the minimal amount of space required for representing all the particles, plus a fixed memory overhead of the form: $\mathsf{nbCores} \times \mathsf{nbCells} \times \mathsf{sizeOfChunk} \times \mathsf{bytesPerParticle}$, where $\mathsf{bytesPerParticle}$ is 24. For example, in a simulation on a 128x128 grid, with chunks of size 512, executing on 18 cores, the memory overhead is 7.3 GB. This may be significant in absolute terms, nevertheless it is much less than what is required by competing algorithms whose memory overheads are proportional to the number of particles, e.g. accounting for 50% of the total memory usage.

Parameters

N: number of particles.
$X \times Y$: size of the grid.
Δt: duration of a time step.

Variables

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

Algorithm

```
Foreach time step
    Set all cells of $\rho$ to 0
    Foreach particle
        Read $E$ values near particle position
        Update particle velocity                          $v \mathrel{+}= \frac{q}{m} E \Delta t$
        Update particle position                          $x \mathrel{+}= v \Delta t$
        Add particle charge to $\rho$ near particle position
    Compute $E$ from $\rho$                               Poisson solver
```

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

2 PIC Method and Related Work

Fig. 1 shows the general pattern of the PIC method, applied to the resolution of the Vlasov-Poisson system of differential equations shown below, which models the time evolution of the distribution function f of charged particles in a plasma.

$$\begin{cases} \partial_t f + v \cdot \nabla_x f + \dfrac{q}{m} E \cdot \nabla_v f = 0 & \text{Vlasov} \\ \nabla_x E = \rho = q \left(\displaystyle\int f(x, v, t) dv - 1 \right) & \text{Poisson} \end{cases}$$

In a concrete implementation, one needs to select a particular interpolation scheme for computing the electric field and accumulating the charges. Our code performs linear interpolation from the four corners of the grid cell where the particle lies—the so-called Cloud-in-Cell model [5]. Remark: optimized PIC codes implement the accumulation of the charge into ρ using an intermediate data structure that enables vectorized processing of the four corners [25].

One central aspect in the design of a PIC implementation is how the particles are stored in the shared memory, and how the particles are assigned to the various cores acting over this shared memory. A first approach is to represent each 2d particle with 32 bytes (4 `doubles`) to describe their positions and velocities. A more efficient approach is the "index plus offset" representation [8, III.E.]. The idea is to store the index of the containing cell (1 `int`, 4 bytes) and the position of the particle relative to the corner of that cell (2 `floats`, 8 bytes). This representation requires 28 bytes per particle if stored in an SoA fashion, but 32 bytes per particle if stored in an AoS fashion, due to padding. In the remainder of this paper, we will assume that every algorithm uses this representation.

A common approach consists of storing the particles in a static array. Prior work has investigated the benefits of sorting this array by cells, to improve locality [7, 17]. Sorting may be performed either in between every iteration, or only every so many iterations. Note that the best frequency for sorting is not so easy to select: it is both architecture-dependent (due to the relative benefits of locality) and input-dependent (particles move faster in a "hot" plasma). Even when sorting is involved, the array of particles may be stored either in an Array of Structures (AoS) [8] fashion, or in a Structure of Arrays (SoA) [3] fashion.

Going further in terms of sorting, one may try to keep the particles sorted by cell at all times. In other words, instead of storing particles directly in an array, one stores the particles in `nbCells` distinct sets of particles. This approach has two main benefits: locality is exploited at its best, and only 24 bytes are required per particle as there is no need to store the cell index. The key challenge is how to represent sets of particles, given than the size of these sets may vary dynamically as the particles move across the grid.

In the Particle-Particle/Particle-Mesh algorithm [16, Sect. 8.4.], each set is represented as a linked list. Yet, this data structure is very inefficient due to memory indirections. Alternatively, one could use a vector (resizable array). However, the copy involved in the resize operations, despite their $O(1)$ amortized cost, induce a significant slowdown in practice: using `std::vector` from `C++` in simulations with an average of $2,288$ particles per vector incurred a 50% slowdown compared to our chunks. Another approach is to "hope" that the distribution of particles does not become very unbalanced, at least no more than by some constant factor (e.g. 2). Under this assumption, one may represent each set as a fixed-size array. The resulting representation is an Array of Arrays of Structures (AoAoS) [24, 10]. The arrays have their size fixed at the beginning of the simulation. If, at some point in the simulation, the number of particles in a given cell exceeds this size, an error is triggered and the simulation must be interrupted. This approach is thus not very robust.

Other researchers have investigated more evolved dynamic set data structures, combining arrays with trees, such as the Packed Memory Arrays (PMA) [4, 12]. This structure consists of a big array containing a fraction of unused cells, and that supports dynamic rebalancing of these "holes". Yet, dealing with the holes and rebalancing them increases the number of memory operations, resulting in poorer performance. Furthermore, the parallelization scheme proposed for PMA [11, Chap. 5] incurs additional overheads, as the structure then needs to be scanned twice. Particle binning [20] is a closely related technique that can be efficiently parallelized. However, its efficiency critically relies on the assumption that only a small fraction (e.g. 2%) of the particles change cell at each time step.

One closely related piece of work [9] targets GPU hardware and is based on *frame lists*, a structure analogous to our chunks. This work nevertheless differs from ours in two major ways. First, it stores particles by supercells (blocks of adjacent cells), whereas we organize them by cell. We thereby save the need to store the cell index of each particle. Second, this prior work updates in place the particles that do not change supercell but move other particles to their correct frame list using atomic operations. This process leaves holes that are removed in a subsequent compaction pass. In contrast, we require a single pass over the particles, and we avoid the need for atomic operations.

Fig. 2 summarizes the memory usage of the aforementioned algorithms, to compare against our proposal, which, asymptotically, requires a smaller amount of memory. The last column shows that, for 64 GB of total memory or more, our algorithm is able to fit a much larger number of particles in memory.

2d Particle-in-Cell multicore algorithm	Memory usage (in bytes)[1]	Largest N for 64 GB (in billions)
Out-of-place counting sort (AoS) [8]	$32 \cdot 2N$	0.9
Out-of-place counting sort (SoA) [3]	$28 \cdot 2N$	1.0
Always sorted, static arrays (AoAoS) [24]	$\geq 24 \cdot 1.5 \cdot N$	≤ 1.6
Always sorted, packed arrays [12, 11]	$24 \cdot (1.4N + M)$	$1.0 \leq N \leq 1.7$
In-place counting sort (AoS) [7]	$32 \cdot N$	1.8
Buffered counting sort (SoA) [17]	$28 \cdot (N + M)$	$1.0 \leq N \leq 2.0$
Always sorted, binning (AoSoA) [20]	$24 \cdot 1.17 \cdot N$	2.0
Always sorted, chunk bags (this paper)	$\left(24 + \frac{16}{\mathsf{chunkSize}}\right) \cdot N + C$	2.1

Fig. 2. Memory usage of 2d PIC implementations for multicores. N denotes the number of particles, M is the maximum number of particles crossing cell boundaries on one iteration (M can be up to N in our simulations), and $C = 24 \times \mathsf{chunkSize} \times \mathsf{nbCores} \times (2\,\mathsf{nbCells} + 1) + \mathcal{O}(\mathsf{nbCells} \times \mathsf{nbCores})$, which is a constant for a given grid and hardware.

3 Our Multicore Algorithm for the PIC Method

Our approach is based on a realization of the sets of particles using a data structure, which we here refer to as *chunk bag.* This data structure is an optimized variant of a relatively standard structure for representing extensible sequences. A chunk bag essentially consists of a linked list of fixed-capacity arrays, called *chunks.* Each chunk stores a pointer to the next chunk (possibly a null pointer), a fixed-capacity array of particles, and a size field. Each bag stores a pointer on its first chunk and on its last chunk from that linked list.

As an optimization, a bag also keeps pointers to the next available location in the array of the last chunk, and to the location one past the last in the last chunk. These auxiliary pointers save an indirection each time we add a particle to the data structure—such optimizations are typical for container data structures [15]. As an exception, we do not maintain the size field of the back chunk, since this size value can be deduced from the two auxiliary pointers. In summary:

```
struct chunk { struct chunk* next; int size;
               particle items[CHUNK_SIZE]; } chunk;
struct { chunk* front, back; particle* back_end, back_head; } bag;
```

The bag data structure supports the following operations. **Add**: inserts a particle into a bag, in $\mathcal{O}(1)$. An insertion may require allocating a new chunk, but the associated overhead is amortized over the size of a chunk. Moreover, since all chunks have the same size, allocation and deallocation are optimized using free lists. **Iter**: iterates over all the particles in the bag. This operation is almost as efficient as iterating over a static array. Most importantly, chunks

[1] In [24], the factor 1.5 allows each cell to contain up to 50% more particles than the average; above that threshold, the simulation must be interrupted. In [12], the factor 1.4 comes from the fact that 40% of the array is reserved for unused cells (holes). In [20], the factor 1.17 similarly corresponds to 6% unused cells and overflow buffers. In our work, the term $\frac{16}{\mathsf{chunkSize}}$ accounts for the size of the fields `next` and `size` associated with each chunk (the computation of our largest N uses $\mathsf{chunkSize} = 512$).

```
bag particles$[0..\mathsf{nbCells}-1]$, particlesNext$[0..\mathsf{nbCores}-1][0..\mathsf{nbCells}-1]$
double $\rho[0..X][0..Y], E[0..X][0..Y], \rho$Next$[0..\mathsf{nbCores}-1][0..\mathsf{nbCells}-1][0..3]$
1  Foreach time step
2    Set in parallel
3      particlesNext$[0..\mathsf{nbCores}-1][0..\mathsf{nbCells}-1]$ to empty
4      $\rho$Next$[0..\mathsf{nbCores}-1][0..\mathsf{nbCells}-1][0..3]$ and $\rho[0..X][0..Y]$ to zero
5    Parallel Foreach idCell in $0\ldots$ nbCells-1
6      Read $E[x][y]$, for each $(x,y)$ among the 4 corners of cell idCell
7      Foreach chunk in particles[idCell]
8        Foreach particle in that chunk
9          Update particle velocity
10       Foreach particle in that chunk
11         Update particle position
12         Compute idCellNext, the index of the cell containing the particle
13         Add the particle into particlesNext[currentCoreId][idCellNext]
14         Accumulate its charge into $\rho$Next[currentCoreId][idCellNext][0..3]
15       Deallocate that chunk
16   Parallel Foreach idCell in $0\ldots\mathsf{nbCells}-1$
17     Set particles[idCell] to particlesNext[0][idCell]
18     For idCore in $1\ldots$ nbCores-1
19       Merge particlesNext[idCore][idCell] into particles[idCell]
20     For idCore in $0\ldots$ nbCores-1, For $i$ in $0\ldots3$
21       $\rho[x][y]$ += $\rho$Next[idCore][idCell][$i$], where $(x,y)$ is $i$-th corner of cell idCell
22   Compute $E$ from $\rho$ using a Poisson solver
```

Fig. 3. Our parallel algorithm for the PIC method on multicore architectures.

may be deallocated while the iteration over the bag proceeds (line 15). **Merge**: two bags may be merged in-place, in $\mathcal{O}(1)$, by concatenating the two linked lists involved. Importantly, no compaction is involved. In particular, after a merge, a non-full chunk may appear in the middle of a linked list of chunks.

The pseudo-code of our algorithm appears in Fig. 3. The key ideas have been described in the introduction. An important addition is the loop fission that we have applied in order to exploit the Single Instruction on Multiple Data (SIMD) feature. Particles update their velocity by interpolating the value of the electric field at their position. Since the interpolation formula is the same for all particles from a same cell, it may be implemented using vectorized instructions. To that end, we isolated the velocity update operations. As long as the data from one chunk fits into the L1 cache, this does not increase the number of accesses to the main memory. Otherwise, an additional level of tiling can be applied.

To summarize, our algorithm has three key features. First, at each step, each particle is read from and written into the main memory exactly once (read on line 9, still in cache for lines 11-14, and write on line 13). Thus, our algorithm does not perform unnecessary accesses to the main memory. Second, each time step involves only three synchronization points: one at the end of each parallel loop (lines 2, 5, and 16). Third, thanks to the use of core-indexed data structures for ρ and for particlesNext, we avoid data races and do not need atomic operations.

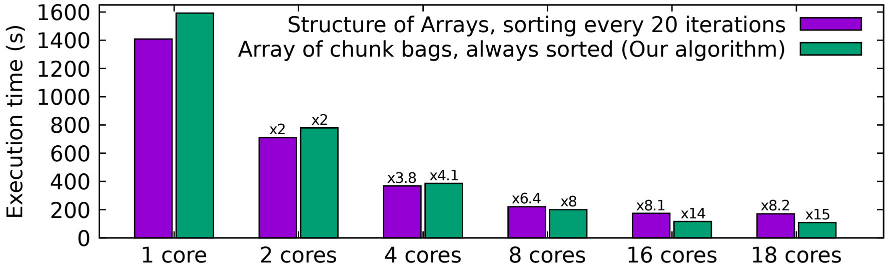

Fig. 4. Strong scaling: 100 iterations on a 128x128 grid with 900 million particles.

4 Empirical Results

Our experiments were conducted on the Marconi supercomputer, on which we were granted the use of 64 nodes with 2 sockets each. Each socket is an Intel Xeon E5-2697 v4 @2.3 GHz (Broadwell), with 64 GB of RAM, 4 memory channels, and 18 cores. Our `C` code was compiled using Intel C Compiler 17.0.1, using the FFTW3 library [13] for the Poisson solver, and storing 512 particles per chunk.

We ran simulations on two classical test cases [6, 16] and checked that they matched the expected mathematical results. We used periodic boundary conditions, and the following initial distributions:

$$\left(1 + 0.01 \cos\left(\frac{x}{2}\right) \cos\left(\frac{y}{2}\right)\right) \frac{1}{2\pi v_{th}^2} \exp\left(-\frac{v_x^2+v_y^2}{2v_{th}^2}\right) \qquad \text{Landau damping}$$

$$\left(1 + 0.1 \left(\cos\left(\frac{y}{2}\right) + \cos\left(\frac{x+y}{2}\right)\right)\right) \frac{v_x^2}{2\pi v_{th}^2} \exp\left(-\frac{v_x^2+v_y^2}{2v_{th}^2}\right) \qquad \text{Two-stream instability}$$

One important challenge faced by prior work is that performance significantly depends on the percentage of particles crossing cell boundaries at each time step. In constrast, the performance of our algorithm should, by design, not depend so much on the percentage of crossing particles. To empirically verify this claim, we increased particle velocities by a factor 100 (raising v_{th} from 0.01 to 1.0). For Landau damping, this increased the percentage of crossing particles from 1.8% to 87%, but increased execution time by only 4.64%. For two-stream instability, this increased the percentage of crossing particles from 12% to 98%, but increased execution time by only 4.59%.

Figure 4 reports a strong scaling for our algorithm, and compares it with prior work using SoA [3], carefully optimized for the same architecture. Although the SoA algorithm is slightly faster when using 4 cores or less, our algorithm, which puts less pressure on the memory bus, outperforms it for more cores. With 18 cores, our algorithm is 36% faster and is able to update 861 million particles per second. Note that this experiment simulates 900 million particles, which is the maximum that out-of-place sorting can accommodate, whereas our algorithm could handle more than twice as many particles.

Figure 5 shows the memory bandwidth of our code when performing a weak scaling. We take as reference the Stream benchmark [19], which aims at evaluating the maximal bandwidth that can be reached in practice. The Stream bench-

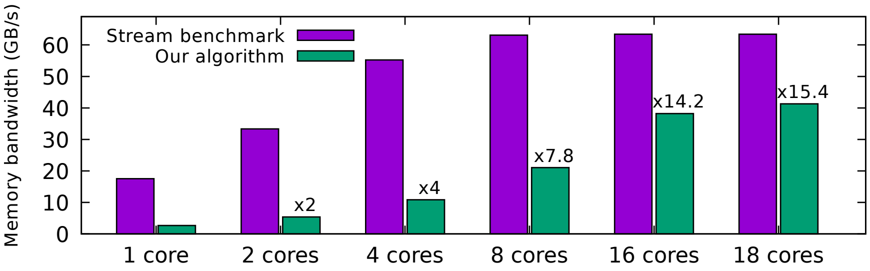

Fig. 5. Memory bandwidth: 100 iterations on a 128x128 grid with 100 million particles per core (up to 1.8 billion particles in total). Bandwidth is measured as: nbIterations × nbParticles × sizeof(particle) × 2 / executionTime. The actual bandwidth may be even slightly higher, as our count does not include chunk management operations.

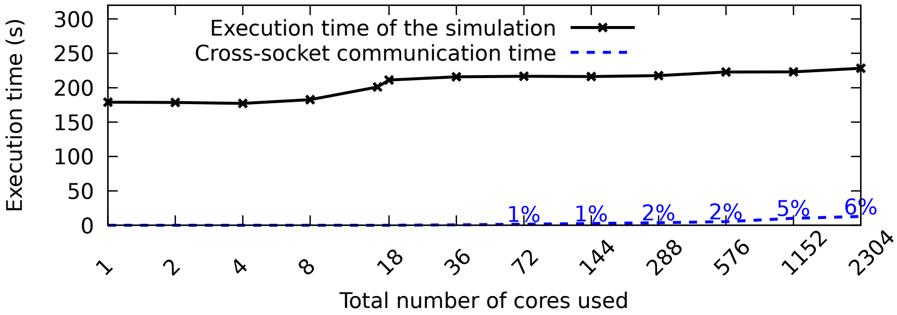

Fig. 6. Weak scaling: 100 iterations on a 128x128 grid with 100 million particles per core (up to 230 billion particles in total), on up to 128 18-core sockets.

mark reaches 63.4 GB/s, which corresponds to 83% of the theoretical peak of our hardware (76.8 GB/s). As Figure 5 shows, on 18 cores, our algorithm reaches more than 65% of the reference memory bandwidth. Since our algorithm does not perform unnecessary accesses to the main memory, we conclude that our code is not far from exploiting the machine at its best.

Figure 6 reports on the performance of hybrid parallelism, with a weak scaling of our code on 128 sockets (2,304 cores), using one MPI process per socket, and 18 OpenMP threads per socket, i.e. one thread per core. The results show an almost perfect scaling, with only 8% overhead when scaling from 1 to 128 sockets. This overhead is expected, due to the (logarithmic) communication costs involved in the `MPI_ALLREDUCE` communication. This experiment demonstrates the efficiency of our parallel algorithm at the scale of 230 billion particles.

Acknowledgments: 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] M. Bader. *Space-Filling Curves*. Springer-Verlag Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-31046-1.

[2] J. Barnes and P. Hut. “A hierarchical O(N log N) force-calculation algorithm”. In: *Nature* 324.3 (1986), pp. 446–449. DOI: 10.1038/324446a0.

[3] Y. Barsamian, S. A. Hirstoaga, and É. Violard. “Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell Code”. In: *2017 IEEE Intl. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*. 2017, pp. 1168–1177. DOI: 10.1109/IPDPSW.2017.74.

[4] M. A. Bender, E. D. Demaine, and M. Farach-Colton. “Cache-oblivious B-trees”. In: *Proc. of the 41st Annual Symp. on Foundations of Computer Science (FOCS)*. 2000, pp. 399–409. DOI: 10.1137/S0097539701389956.

[5] C. K. Birdsall and D. Fuss. “Clouds-in-Clouds, Clouds-in-Cells Physics for Many-Body Plasma Simulation”. In: *J. Comput. Phys.* 3 (1969), pp. 494–511. DOI: 10.1006/jcph.1997.5723.

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

[7] K. J. Bowers. “Accelerating a Particle-in-Cell Simulation Using a Hybrid Counting Sort”. In: *J. Comput. Phys.* 173.2 (2001), pp. 393–411. DOI: 10.1006/jcph.2001.6851.

[8] 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.

[9] 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: *Proceedings of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. 2013, 5:1–5:12. DOI: 10.1145/2503210.2504564.

[10] 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.

[11] M. Durand. “PaVo. An Adaptative Parallel Sorting Algorithm.” PhD thesis. Université de Grenoble, 2013.

[12] 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.

[13] 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.

[14] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, and A. Bhattacharjee. “The Plasma Simulation Code: A mod-

ern 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.

[15] D. R. Hanson. “Fast Allocation and Deallocation of Memory Based on Object Lifetimes”. In: *Softw. Pract. Exper.* 20.1 (1990), pp. 5–12. DOI: 10.1002/spe.4380200104.

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

[17] 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.

[18] C. C. Kim and S. E. Parker. “Massively Parallel Three-Dimensional Toroidal Gyrokinetic Flux-Tube Turbulence Simulation”. In: *J. Comput. Phys.* 161.2 (2000), pp. 589–604. DOI: 10.1006/jcph.2000.6518.

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

[20] 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.

[21] D. M. Nicol. “Rectilinear Partitioning of Irregular Data Parallel Computations”. In: *J. Parallel Distr. Com.* 23.2 (1994), pp. 119–134. DOI: 10.1006/jpdc.1994.1126.

[22] I. Surmin, A. Bashinov, S. Bastrakov, E. Efimenko, A. Gonoskov, and I. Meyerov. “Dynamic Load Balancing Based on Rectilinear Partitioning in Particle-in-Cell Plasma Simulation”. In: *Parallel Computing Technologies: 13th Intl. Conf. (PaCT).* 2015, pp. 107–119. DOI: 10.1007/978-3-319-21909-7_12.

[23] X. Sáez, A. Soba, E. Sánchez, R. Kleiber, F. Castejón, and J. M. Cela. “Improvements of the particle-in-cell code EUTERPE for petascaling machines”. In: *Comput. Phys. Commun.* 182.9 (2011), pp. 2047–2051. DOI: 10.1016/j.cpc.2010.12.038.

[24] 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.

[25] 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.

[26] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. “A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations”. In: *Comput. Phys. Commun.* 183.4 (2012), pp. 880–889. DOI: 10.1016/j.cpc.2011.12.013.