EUROFUSION WP14ER-PR(16) 16091

E Lanti et al.

# A Portable Platform for Accelerated PIC Codes and Its Application to Multicore and Many Integrated Cores Architectures Using MPI and OpenMP

Preprint of Paper to be submitted for publication in
Computer Physics Communications

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. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

# A Portable Test Bed for Accelerated PIC Codes and its Optimization on Multicore and Many Integrated Cores Architectures Using MPI and OpenMP

E. Lanti[a,*], T. M. Tran[a], A. Jocksch[b], F. Hariri[c], S. Brunner[a], C. Gheller[b], L. Villard[a]

[a] *École Polytechnique Fédérale de Lausanne (EPFL), Swiss Plasma Center (SPC), CH-1015 Lausanne, Switzerland*
[b] *CSCS, Swiss National Supercomputing Centre, Via Trevano 131, 6900 Lugano, Switzerland*
[c] *CERN, CH-1211 Geneva 23, Switzerland*

**Abstract**

With the aim of developing efficient optimizations and parallelization algorithms for porting application PIC codes to new multi- and manycore HPC platforms, a test bed called the `pic_engine` has been developed as a simple and portable abstraction of the PIC algorithm. In this work, we use it to test the hybrid MPI/OpenMP implementation as well as different algorithmic optimizations. Among them, a counting sort method is implemented to increase data locality in the memory and thus increase the performance. Thanks to the portability of the `pic_engine`, several optimization algorithms are tested on the two Cray XC30 and XC40 machines from CSCS in Switzerland and the Bullx B510 supercomputer at IFERC-CSC in Japan equipped with both CPUs and Intel Xeon Phi processors. The results show that with our optimizations the performance can be increased by at least a factor 2.15 compared to the initial non-optimized version of the `pic_engine`.

*Keywords:* Particle-In-Cell, parallelization, OpenMP, MPI, multithreading, vectorization

## 1. Introduction

Particle-In-Cell (PIC) codes are widely used in computer simulations as a mean to solve integro-differential equations. Developed in the mid 50's for hydrodynamic problems [1], the PIC algorithm has quickly gained popularity in the plasma physics community [2, 3] and is now extensively used to solve the Vlasov-Maxwell (or Fokker-Planck) problem [4, 5, 6, 7].

In the PIC algorithm, so-called numerical particles, or markers, are used to represent the physical particle's distribution function and are evolved in their continuous phase space in a Lagrangian frame. Typically, a PIC simulation consists in a sequence of discrete time steps for each of which the marker equations of motion have to be solved. To that end, the contribution from each marker to the electric charge and current fields are first deposited on a discretized grid and the self-consistent electromagnetic fields are computed. Then, the Lorentz forces acting on the markers are evaluated by interpolating the electromagnetic fields at the particle's positions which enables to advance their velocities and finally update their positions.

*Corresponding author: E. Lanti; Email: emmanuel.lanti@epfl.ch

To resolve the high-dimensional phase space, in any case, requires significant numerical resources. Furthermore, based on the statistical Monte-Carlo method, the PIC algorithm generally requires a large amount of numerical markers to reduce the statistical noise. Thus, PIC codes usually need the use of High Performance Computing (HPC) platforms that provide the required large computing capacities.

During the last two decades, PIC codes have dramatically evolved to take full advantage of the low-level parallelism provided by the present HPC clusters, usually using the Message Passing Interface (MPI) programming model. However, recently emerging platforms equipped with multi- and manycore shared memory processors now supply an additional level of parallelism through threads that has to be exploited.

In this paper, we present various optimization techniques as well as parallelization algorithms that were tested using the `pic_engine`. This platform has been developed as a test bed allowing one to easily implement and test algorithms in a framework retaining only the key elements of the PIC algorithm. It is designed to be modular and portable such that it can be used on different multi- and manycore architectures such as GPU-equipped machines [8, 9, 10]. Here, this study focuses on multicore CPUs and the Intel Xeon Phi Many Integrated Core (MIC) [11] processors. The portability is an important aspect of the work since the `pic_engine` is meant, in a future work, to deploy generic optimizations to the PIC algorithm that will be used to port to the multi- and manycore platforms full application codes such as the global gyrokinetic code ORB5 [6, 12] used to simulate turbulence in magnetically confined plasmas and the RAMSES [13] code that models astrophysical systems. To this end, the `pic_engine` is designed to be the simplest abstraction possible of the PIC algorithm but nevertheless retaining its essential elements. In this way, we can study the optimization techniques implemented on the most general parts of the algorithm and treat the fundamental parallelization problems inherent to the PIC method.

As a base case for this work, the `pic_engine` evolves a simple plasma physics problem and the methods will thus be designed with this goal in mind. However, the portability is ensured by only treating the elements of the PIC algorithm that are common between different research fields.

The `pic_engine` is written in Fortran using hybrid MPI and OpenMP for the CPU parallelization. The OpenMP API has been chosen as it is standardized and portable. Furthermore, it is usually easy to handle and can be implemented incrementally. Note that with this programming model, the same source code can be run either on CPUs or MICs in native or symmetric modes, i.e. treating the MICs as standalone processors.

The paper is organized as follows. In section 2, we present the `pic_engine` platform, its main methods, and how it is parallelized. Section 3 is the main section of the paper presenting the performance analysis of the `pic_engine` components as well as the overall performance of the program for both single and multinode tests. Finally, the conclusions are given in section 4.

## 2. The `pic_engine` platform

Particle-In-Cell codes are widely used in plasma physics applications as a mean to evolve the plasma species distributions according to the kinetic Vlasov-Maxwell system of equations. The physical particles are represented

by a set of numerical markers that are continuously evolved in phase space and represent the sources for the self-consistent electromagnetic fields. The basic PIC algorithm adopts a Lagrangian description of the system which is evolved following four steps, as seen in Figure 1. The first one is the charge assignment, called `setrho`, in which the charge and current carried by the particles are deposited from the particle's positions ($p$ index) to the field grid ($i$ index). In the second step, the self-consistent electromagnetic fields $\vec{E}$ and $\vec{B}$ are computed on the grid according to Maxwell's equations and for the charge and current deposited previously. In the following step, called `accel`, the Lorentz force acting on the particles is evaluated and interpolated back from the field grid to the particle's positions. Finally, the particle's positions and velocities are updated in the `push` step. A PIC simulation therefore typically consists of a set of time iterations during which each of these four steps is carried out.

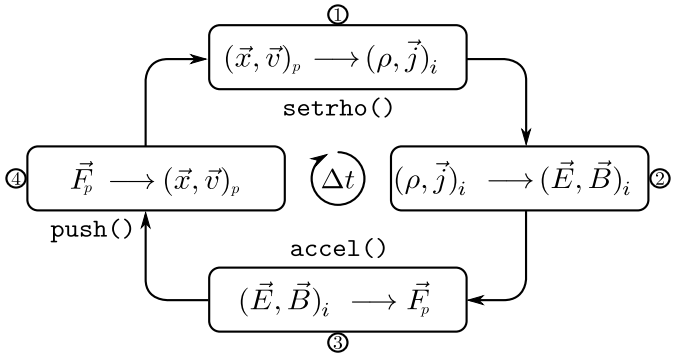


Figure 1: Time loop of the PIC algorithm. Each iteration consists of four steps. (1): The contribution to the charge and current fields are deposited from the particle's positions ($p$ index) to the field grid ($i$ index). In the `pic_engine`, this step is called `setrho` and only the charge is deposited. (2): Using the deposited charge and current, the electromagnetic fields $\vec{E}$ and $\vec{B}$ are computed by the field solver. (3): The Lorentz forces acting on the particles are estimated based on the electromagnetic fields which are interpolated from the grid to the particle's positions. These forces allow updating the particle's velocities. (4): Finally, the particle's positions are updated. In the `pic_engine`, the last two steps are called `accel` and `push` respectively.

### *2.1. Structure of the* `pic_engine`

With the aim of developing and testing optimization techniques and parallelization schemes for porting PIC codes to the new multicore and MIC architectures, the `pic_engine` has been implemented using Fortran and parallelized with OpenMP and MPI. It corresponds to an abstraction of the PIC algorithm presented in the previous paragraph, retaining only its key elements. For the sake of simplicity, it evolves a single-species plasma in the electrostatic limit. No magnetic field is considered and the electric field $\vec{E}(\vec{x})$ is imposed and assumed periodic in all three configuration space dimensions and stationary in time:

$$\varphi(x,y,z) = \cos(k_x x)\cos(k_y y)\cos(k_z z)\,, \tag{1}$$

$$\vec{E}(x,y,z) = -\nabla\varphi(x,y,z), \tag{2}$$

where $\varphi(x, y, z)$ is the electrostatic potential and $k_i$ are the components of an arbitrary wave vector. Consequently, no field solver is required here. Note that the computation of the fields reduces to a linear problem that can be solved with various types of approaches, e.g. direct, iterative and multigrid methods. Since these methods and their parallelization are well known, they are not considered here. Due to the electrostatic approximation, only the charge is deposited on the field grid in the charge assignment step `setrho`. In the `accel` step, to evaluate the Lorentz force $\vec{F}$, the electrostatic field is interpolated from the field grid to the particle's positions which then allows evolving the particle's velocities. Similarly, the positions of the particles are updated in the `push` method. In these last two steps, the time integration is done using a second order leap-frog scheme that is energy conserving and reversible [2], see Section 2.1.3 for more details.

At this point, it is important to note that the `pic_engine` is only meant to study the algorithmic performance of the different parts of the PIC algorithm and no physical results are sought. Thus, the absence of a field solver and the use of the aforementioned approximations are not critical for our study. Furthermore, with a field solver and some minor changes, the `pic_engine` can solve different plasma physics problems of interest [14]. Also, we keep in mind that for application codes such as ORB5 using non-standard PIC methods, e.g. involving higher order schemes or more complex equations of motion, the numerical cost of `accel`, `push` and `setrho` is greatly enhanced by the complexity of the problem.

*2.1.1. Data structure and particle initialization*

In the `pic_engine`, field and particle data structures have to be distinguished. The field quantities, in particular, the electrostatic field and the particle charge density are represented on a 3D periodic domain of length `lx`, `ly` and `lz` discretized in `nx`, `ny` and `nz` intervals. The subscripts represent along which direction of the Cartesian coordinates $(x, y, z)$ the quantity applies. On the other hand, particle quantities such as their positions and velocities can take any values in the phase space $(\vec{x}, \vec{v})$ and are stored in the `part_att(natts,np)` array, where `natts` represents the six particle attributes, namely the Cartesian coordinates of the positions and the corresponding three coordinates of the velocities, and `np` is the number of particles in the system. The `part_att` array can be stored either in Array Of Structures (AOS), i.e. `part_att(natts,np)` or in Structure Of Arrays (SOA), i.e. `part_att(np,natts)`. As we shall see in the next sections the way of storing data in memory significantly affects the performance.

At the beginning of the simulation, the particle's positions and velocities are initialized using the Fortran `random_number` method. The positions are uniformly distributed in the periodic configuration domain. The velocities $v_x$ and $v_y$ are initially distributed uniformly with $|\mathtt{vx}, \mathtt{vy}| \leq \mathtt{vmax}$, where the parameter `vmax` sets a maximum velocity while $v_z$ can be either uniformly distributed as $v_x$ and $v_y$, or distributed according to a normalized Boltzmann distribution.

*2.1.2. Parallelization*

The `pic_engine` parallelization is achieved using a hybrid MPI/OpenMP implementation, see Figure 2. First, the 3D domain is decomposed into `nsd` subdomains in the $z$ direction. Each subdomain grid data is then replicated into `nclones` clones and this 2D decomposition is mapped onto an MPI grid. Finally, a third level of parallelism is added using OpenMP by assigning `nthread` threads to each MPI task, so that the total number of cores used is equal to the product `nsd` $\times$ `nclones` $\times$ `nthread`. Usually, the number of cores used by the program is equal to the number of physical cores on the processor. However, using Intel's Hyper-Threading technology, it is possible to initialize more threads than physical cores. Depending on the application, this can be beneficial or, in the worst case, can decrease the performance.

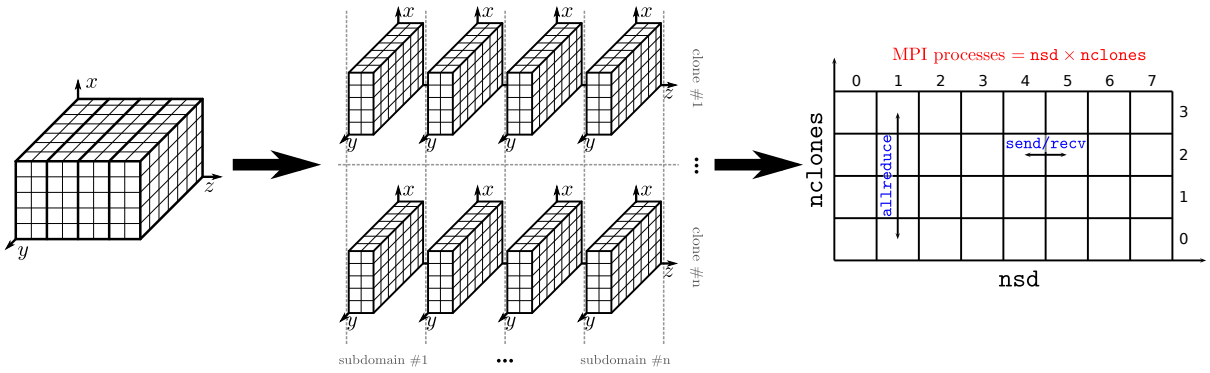


Figure 2: Illustration of the `pic_engine` parallelization. The 3D domain is first decomposed into `nsd` subdomains in the $z$ direction. Each subdomain is then copied `nclones` times and this 2D decomposition is mapped onto an MPI 2D grid. Finally, OpenMP is introduced and `nthread` threads are affected to each clone. With this parallelization scheme, only two types of communications are needed for three operations: a reduction is done across the clones to gather the partial charges with MPI `allreduce` and particles and guard cells are transferred across subdomains with MPI point-to-point `sendrecv`.

It is important to note that communications are only needed for three operations. First, all the partial charges computed by `setrho` on each clone of one subdomain must be added using MPI `allreduce` operations and the guard-cells must be exchanged with neighboring subdomains using MPI point-to-point `sendrecv`. Then, the particles have to be transferred between subdomains; this is done by the `pmovez` routine using MPI point-to-point `sendrecv` directives. The `pmovez` method allows one to transfer the particles not only to the nearest neighboring subdomains but is applicable to an arbitrary source/destination pattern.

*2.1.3. Charge deposition, particle accelerating and pushing*

As we have seen at the beginning of this section, the `pic_engine` consists of three main steps: the charge deposition, the particle accelerating, and pushing. In the program, they are implemented respectively in the `setrho`, `accel` and `push` methods. In this section, we discuss their implementation which is mainly inspired by [2].

The first step in the PIC algorithm is the charge deposition. Its goal is to collect the electric charge density

$\rho$ of all particles and to deposit it on the field grid with the aim of computing the electrostatic field. Thus, for each particle, we have to compute

$$\rho_{grid} \longleftarrow \rho_{grid} + \rho_p^{dep} \text{ (particle to grid)}, \tag{3}$$

where $\rho_{grid}$ is the charge already deposited on the field grid and $\rho_p^{dep}$ is the charge of the current particle deposited on this same grid using a first-order particle weighting. This approach, also called Cloud-In-Cell (CIC), works as follows. For a particle that lies in the interval $[x_i, x_{i+1})$, we first compute its relative position:

$$w_p = \frac{x_p - x_i}{\Delta x},$$

where $x_p$ is the particle's position and $\Delta x$ is the length of the interval. Then, we deposit the charge according to

$$\rho_p^{dep}(x_i) = 1 - w_p, \qquad \rho_p^{dep}(x_{i+1}) = w_p.$$

Note that in the `pic_engine`, the equations are normalized such that the elementary charge $e$ and the mass $m$ of the particles are unity in absolute value. The charge deposition clearly involves a gather operation since particles are distributed randomly in configuration space, resulting in an indirect addressing to the $\rho$ array when depositing the contribution from each particle. This indirect writing has to be taken care of carefully when parallelizing the `setrho` method for shared memory systems to avoid race conditions, i.e. when two or more threads try to modify data stored at the same memory location.

The particle's equations of motion

$$\frac{\mathrm{d}\vec{v}}{\mathrm{dt}} = \vec{F} = -\vec{E}, \tag{4}$$

$$\frac{\mathrm{d}\vec{x}}{\mathrm{dt}} = \vec{v}, \tag{5}$$

are respectively solved by the `accel` and `push` methods. To evolve the particle's positions and velocities, we use the leap-frog method [2] that consists in discretizing the equations with a finite-difference scheme:

$$\frac{\vec{v}_{new} - \vec{v}_{old}}{\Delta t} = \vec{F}_{old}, \tag{6}$$

$$\frac{\vec{x}_{new} - \vec{x}_{old}}{\Delta t} = \vec{v}_{new}, \tag{7}$$

where $F_{old}$ is the Lorentz force computed at $x_{old}$ and $\Delta t$ is the numerical time step. They are then evolved on two separate time-grids staggered by $\Delta t/2$, see Figure 3. By doing so, the leap-frog method is time centered and thus second order accurate with respect to $\Delta t$.

When using this method, two issues must be correctly addressed. First, since the velocities and positions are evolved on two different time grids, care must be taken when computing any quantity involving both the particle's positions and velocities such as the total energy. Secondly, at the beginning of the simulation, both the velocities and positions have to be correctly initialized on their respective time grid.

In the `accel` method, the electrostatic potentials are linearly interpolated from the field grid to the particle's positions resulting in piece-wise linear polynomials in the $x$, $y$ and $z$ directions. The electrostatic field components

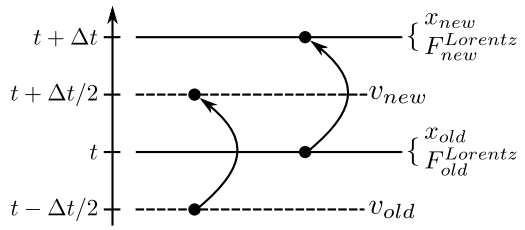


Figure 3: Illustration of the leap-frog method. Newton's law is split in two first order equations: one for the velocities and one for the positions. They are then discretized using a finite-difference scheme and evolved on two time grids staggered by $\Delta t/2$. This scheme is time centered and thus second order accurate.

$E_x$, $E_y$ and $E_z$ that result from the gradient of the potential are, in their turn, piece-wise *constant* in the $x$, $y$ and $z$ directions respectively and piece-wise *linear* in the other two directions. This method is not subject to race conditions because there are no concurrent writes and can, therefore, be parallelized in a straightforward way. However, due to the indirect read of the field array, vectorized operations are not possible because they are not in the instruction set of the processor.

In summary, both `push` and `accel` methods can be trivially parallelized using OpenMP since they consist respectively of vector add and multiply (SAXPY) operations and indirect reads. The most challenging method to parallelize is in fact `setrho`, as it involves indirect writes that are subject to race conditions. For these reasons, we shall particularly focus on `accel` and `setrho` in the remaining of this paper.

### *2.2. Particle sorting*

In the `pic_engine`, and more generally in most PIC codes, the particles are randomly distributed in space. As a consequence, data locality is poor and performance drops due to non-contiguous memory writings in `setrho` and accesses in `accel`. Indeed, cache memories are designed to be efficient with consecutive data accesses. If this is not ensured, significant overhead time will be spent by reloading the data from the main memory. With increased data locality, it is more likely that the data already in the cache can be reused for further computations thus avoiding cache misses and reloading.

To this end, we have implemented a sorting method based on the counting sort algorithm. As a reminder, the algorithm sorts the particles into buckets that typically represent one or more grid cells. This is done in three main steps, see Algorithm 1:

1. In this first step, we create a histogram of the `np` particles by parsing them and computing the indices of their recipient bucket. To this end, the 2D $(x, y)$ plane is mapped on a 1D vector using a linear mapping, i.e.

$$index = \left\lfloor \frac{x_p}{\Delta x} \right\rfloor + \left\lfloor \frac{y_p}{\Delta y} \right\rfloor nx, \tag{8}$$

where $\Delta x$, $\Delta y$, $x_p$ and $y_p$ are the interval lengths and the particle's positions in the $x$ and $y$ directions respectively. Note that we consider only domains with one grid interval in the $z$ direction but a 3D

generalization is straightforward. Other options like Hilbert filling curves [15] are also of interest to preserve data locality across the mapping but they are not considered here.

2. In the second step, an inclusive prefix sum is made. It essentially consists in parsing the `nbtot` buckets to compute the *ending* position of each bucket in the sorted array.
3. Finally, using the results of the prefix sum, the particles are put in their sorted place.
4. The temporary sorted array is copied back to `part_att`.

**Algorithm 1** The counting sort algorithm is composed of three main loops that respectively compute the histogram of the particles in the buckets (lines 1 to 4), compute the starting and ending indices of each bucket with a prefix sum (lines 5 to 7) and move the particles in their correct order with the help of a temporary array (lines 8 to 12).

1: **for** $ip = 1, np$ **do**
2: $id = index(ip)$ ▷ The *index* function returns the particle grid-cell index
3: $count(id) = count(id) + 1$
4: **end for**
5: $displs(1) = count(1)$
6: **for** $ib = 2, nbtot$ **do**
7: $displs(ib) = displs(ib - 1) + count(ib - 1)$
8: **end for**
9: **for** $ip = np, 1, -1$ **do**
10: $id = index(ip)$
11: $part\_att\_tmp(:, displs(id)) = part\_att(:, ip)$
12: $displs(id) = displs(id) - 1$
13: **end for**

Note that the fourth step was added because this algorithm is out-of-place and hence requires a temporary array. Memory can be saved with in-place versions but such methods [16] have not been implemented because memory was not an issue for this application.

A straightforward algorithmic analysis shows that the timing of this sorting is linear in the number of particles and buckets, i.e. $T_{psort} = \mathcal{O}(\texttt{np}) + \mathcal{O}(\texttt{nb})$, where $T_{psort}$ is the serial timing of the sorting method.

*2.2.1. Vectorization*

Both `setrho` and `accel` methods have to read the particle's positions to compute in which grid cell they belong. This requires an indirect addressing in the particle loop which prevents the compiler to use auto-vectorization because no such vectorization operations are in the instruction set of the processor. To solve this issue, we can take advantage of the particle sorting. Indeed, with a full sorting, i.e. if the buckets coincides with the grid cells, all the particle loops with indirect addressing can be replaced by a double loop involving first a loop over

the buckets and a second loop over the particles in each of these buckets, see Algorithm 2. Indeed, by knowing which bucket one is treating, one automatically knows the field grid positions of the particles if they have been fully sorted. By doing so, the inner loop is totally independent of the indirect addressing and the compiler can vectorize it. This method can be beneficial if the inner loop is long enough to allow a good vectorization.

**Algorithm 2** Illustration of the vectorization procedure. In the first method (lines 1 to 4), the grid cell index of each particle is computed and rest of the routine is executed. In the second method (lines 6 to 12), the loop over the particle is decomposed into a loop over the grid cells and a loop over the particles in the grid cells. With this approach, all the particles within each bucket have the same grid cell index for the inner loop. Hence, the compiler can vectorize it.

```
1:  for ip = 1, np do
2:      ! Compute particle grid cell index
3:      ...
4:  end for
5:
6:  for ib = 1, nx * ny * nz do
7:      ! Compute particle grid cell index
8:      for ip = displs(ib) + 1, displs(ib + 1) do
9:          ! Then do the calculation for each particle in the current grid cell
10:         ...
11:     end for
12: end for
```

## 3. Performance analysis

In this section, we present the different optimizations made to the `pic_engine` as well as the timings and performance gains obtained. The study was performed on Piz Daint and Piz Dora from the Swiss National Supercomputing Centre (CSCS) and the Helios supercomputer at the Computational Simulation Center of the International Fusion Energy Research Centre (IFERC-CSC) in Japan. All the relevant characteristics of these machines are summarized in Table 1.

For both the CSCS machines, Piz Daint and Piz Dora, the Cray compiler `cce` 8.3.12 and the Cray MPICH 7.2.2 library were used. On Helios, the Intel compiler 15.0.2 in conjunction with Intel MPI 5.0.3 was used. Except for the MICs that showed a better performance with four threads per core, the hyper-threading was always disabled to guarantee an optimal performance. All the available cores were used by ensuring that the product `nsd` $\times$ `nclones` $\times$ `nthread` was always equal to the total number of cores. In this paper, `nclones` equal to 1 corresponds to a pure OpenMP run and `nclones` equal to the number of cores per node corresponds to a

pure MPI run. Note that when we talk about pure OpenMP runs, there may be several MPI processes for the inter-node communications which is typically the case for multinode runs. This terminology is only used here to emphasize that within each node, all the parallelization is made using OpenMP.

For the MIC nodes, three different run modes are available on Helios: offload, symmetric and native. In this paper, to ensure a homogeneous performance among the processors and avoid complicated interactions between MICs and CPUs, we focus on the latter which consists in using the MICs as standalone nodes. This is possible because each of them is equipped with a Linux micro operating system. In this mode, only the MICs are used, as opposed to the symmetric mode where either the CPUs and MICs are used in a similar way. Finally, the offload mode consists in using the MIC as an accelerator. The host, generally the CPU, starts the application and offloads some data and computations to the MIC.

The standard singlenode problem tested in this study consists in a $512(\texttt{nx}) \times 256(\texttt{ny}) \times 1(\texttt{nz})$ grid with $10^6$ particles similar to [8] for the sake of comparison. When performing a multinode study, we essentially make a weak scaling, i.e. the size of the problem is multiplied by the number of nodes such that each has an equivalent per-node workload to the singlenode case. Whenever activated, the sorting was always full, i.e. we considered 512 and 256 buckets in the $x$ and $y$ directions respectively. All the timings given in this paper are normalized to the standard singlenode problem and are reported in nanoseconds per particle and per time step.

In the following, if not mentioned otherwise, only the results on Helios CPU and MIC nodes are presented for the sake of conciseness. For every conclusion, it has been checked that it also applies on Piz Daint and Piz Dora.

### *3.1. Data structure*

As a first result, the effect of the data structure is studied. To this end, the code is compiled with both SOA and AOS data structures and is run on the Helios machine on both CPU and MIC nodes. The singlenode results for the CPU case are presented in Table 2 for both pure MPI ($\texttt{nclones} = 16$) and pure OpenMP ($\texttt{nclones} = 1$). The SOA data structure is always faster than AOS. Most of the gain comes from the `setrho` routine resulting in an overall time gain of 6.4% for the pure MPI case and 14.4% for pure OpenMP. Similar results are observed on the MIC nodes.

In the remaining of this paper, we have always used the SOA data structure as it is the most efficient option.

### *3.2. Particle sorting*

Different thread-parallel implementations of the sorting algorithm have been designed using OpenMP. Essentially, out of the three loops presented in Algorithm 1, only the prefix scan (second loop) is not parallelized because it is quite complex to do so and it never represents more than 4% of the total timing of the sorting. Here is a brief description of the different versions implemented. More details can be found in Appendix A.1.

**`psort_1`:** The first loop is parallelized using OpenMP `reduction` while the third loop uses the `parallel do` directive in parallel with `atomic` to avoid race conditions.

| | **Piz Daint** (Cray XC30) | **Piz Dora** (Cray XC40) |
|---|---|---|
| **Socket(s)** | 1 | 2 |
| **CPU** | Intel Xeon E5-2670 Sandy Bridge | Intel Xeon E5-2690 v3 Haswell |
| **Frequency** | 2.6 GHz | 2.6 GHz |
| **Cores** | 8/CPU | 12/CPU |

| | **Helios** (CPU) | **Helios** (MIC) |
|---|---|---|
| **Socket(s)** | 2 | 2 |
| **CPU** | Intel Xeon E5-2680 Sandy Bridge | Intel Xeon Phi KNC |
| **Frequency** | 2.7 GHz | 1.3 GHz |
| **Cores** | 8/CPU | 60/MIC |

Table 1: Characteristics of the clusters used for this study. Piz Daint and Piz Dora are two computers from the Swiss Supercomputing Centre (CSCS) and Helios is a magnetic fusion research supercomputer from the Computational Simulation Centre (CSC) in Japan. Note that this last computer is equipped with both CPU and MIC nodes.

| | Pure MPI | | |
|---|---|---|---|
| | AOS | SOA | Gain |
| `push()` | 0.84 | 0.78 | 7.95 % |
| `accel()` | 8.58 | 8.47 | 1.25 % |
| `setrho()` | 9.17 | 8.21 | 11.67% |
| Total | 18.62 | 17.5 | 6.40 % |

| | Pure OpenMP | | |
|---|---|---|---|
| | AOS | SOA | Gain |
| `push()` | 1.43 | 1.40 | 2.00% |
| `accel()` | 4.25 | 4.15 | 2.53% |
| `setrho()` | 5.78 | 4.47 | 29.34% |
| Total | 11.47 | 10.03 | 14.36% |

Table 2: Comparison between the AOS and SOA data structures on Helios (CPU) for the pure MPI and OpenMP cases. All the timings are in $ns/particle/\Delta t$. In both cases, the standard problem with unsorted particles is considered. Similar results are observed for the MIC equipped nodes.

`psort_2`: This implementation slightly differs from Algorithm 1. During the first loop, the histogram is computed as well as the index of each particle in the sorted array and OpenMP `atomic` is used to avoid race conditions. By doing so, the third loop only consists in a simple OpenMP `parallel do` loop.

In Figure 4 (left), we show the timings of the sorting methods for different hybrid runs, i.e. we vary the number of clones and threads such that their product corresponds to the total number of available cores per node.

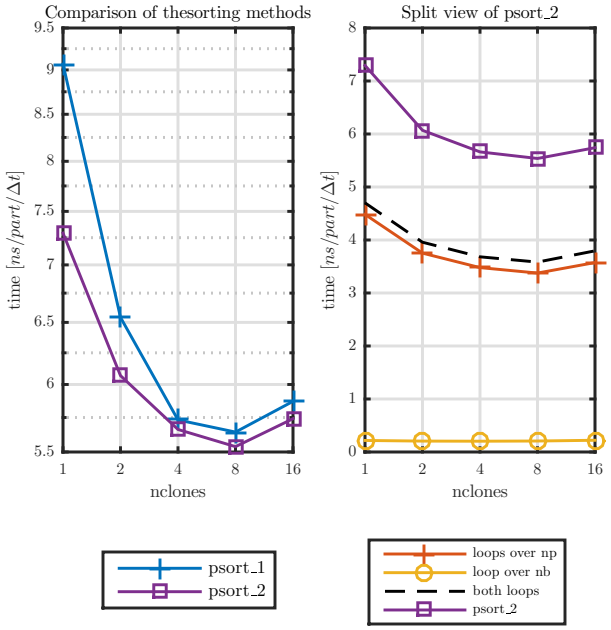


Figure 4: Comparison of the different sorting parallelizations on Helios CPU (left) and split view of `psort_2` showing the timings of the loops composing the sorting algorithm (right). In each case, the timings are shown on a full node such that the product `nclones` $\times$ `nthread` is always equal to the total number of cores.

For the CPU nodes, the timings of the two sorting methods have the same qualitative behavior. For pure OpenMP (`nclones` $= 1$), they are maximum and they decrease towards their minimum value for `nclones` $= 8$ and then, increase for pure MPI (`nclones` $= 16$). To explain this, first recall that the sorting timing depends linearly on both the number of particles and buckets:

$$T_{sort} = T_{np} + T_{nb}, \tag{9}$$

where $T_{np}$ and $T_{nb}$ are two linear functions of the number of particles and buckets respectively. Also, with the decomposition in clones, we have:

$$Np_{clones} \sim \frac{Np_{tot}}{N_{clones}}, \tag{10}$$

where $Np_{clones}$ and $Np_{tot}$ are respectively the number of particles per clone and the total number of particles and $N_{clones}$ is the number of clones. Thus, with an increasing number of clones, each of them must perform the sorting on a fewer number of particles.

In Figure 4 (right) we show the timings of the loop over the buckets (loop over nb) and the two loops over the particles combined (loops over np) composing the sorting method for `psort_2`. First, note that the number of particles sorted per core is constant for the different hybrid runs since the product `nclones` $\times$ `nthread` is constant. This means that $T_{np}$ should be constant. However, for pure OpenMP, we think that the high number of threads tends to saturate the memory bandwidth which explains the timings increase. The loop over the number of buckets is negligible, never more than 4% of the sorting total timing. Finally, the difference between the timing of `psort_2` and the sum of the three loops is due to the copy of the temporary particle array to `part_att`.

In conclusion, method `psort_2` is found to be the most efficient on the CPU nodes. For that reason, it is always used in the following results whenever sorting is activated. The conclusions are equivalent for the MIC nodes.

### *3.3. Charge deposition*

We have seen that both `accel` and `push` are thread-parallelized in a straightforward way, typically with OpenMP parallel loops, whereas care must be taken with `setrho` because of the race conditions furthermore complicated by the indirect writing.

In the `pic_engine`, four different parallelization procedures for the charge deposition are implemented:

**`setrho_1`:** *Threads on particles, collision free, with data replication.* In this method, threads are defined on the particles and we try to mimic OpenMP `reduction`. To this end, race conditions are avoided using private copies of the $\rho$ array for each thread. Although being data safe, this algorithm requires more memory and a scalar reduction must be made at the end of the routine to collect all the data from the threads.

**`setrho_2`:** *Threads on particles, collision resolving, no data replication.* The OpenMP `atomic` directive is used to resolve the race conditions.

**`setrho_3`:** *Threads on particles, collision free, with data replication.* Similarly to the first method, we use the OpenMP `reduction` directive to avoid race conditions.

**`setrho_4`:** *Threads on grid points, collision free, no data replication.* In this method, we use the finite support of the linear CIC method to compute the charge and avoid race conditions. Indeed, with this scheme, a particle will only deposit its charge to the eight nearest grid points (for a 3D problem) so the algorithm consists basically in parsing first the grid points and then the particles contained in the eight nearest grid cells. Note that this method requires the particles to be fully sorted according to their positions.

More details on these parallelizations can be found in Appendix A.2.

In Figure 5, we show the `setrho` timings for the four implementations, for unsorted and sorted particles, without and with vectorization, on a single Helios node. Both CPU (left) and MIC (right) results are shown.

Beginning with the CPU case with unsorted particles, we see that `setrho_1` is the fastest with a best timing of 4.8 ns per particle and per timestep in the pure OpenMP case. This represents a gain of around 2.8 as compared

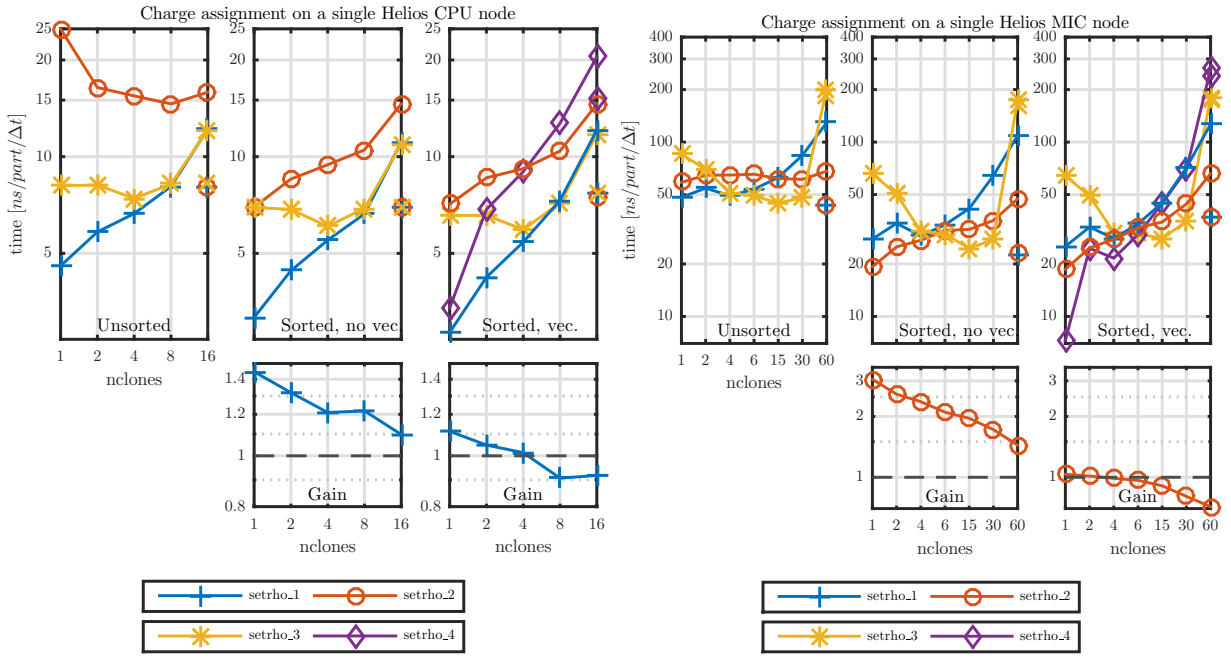


Figure 5: Comparison of the different charge assignment methods on a single CPU (left) and MIC (right) Helios node. Unsorted and both sorted without and with vectorization cases are considered. Method `setrho_4` is only present in the sorted and vectorized case because it requires a full sorting and it is intrinsically vectorized. For both CPU and MIC, the center gain plots show the performance gain of the sorted version of respectively `setrho_1` and `setrho_2` against the unsorted. Similarly, the right gain plots show the performance gain of the vectorized version of respectively `setrho_1` and `setrho_2` compared to the sorted version. The unconnected data points (`nclones` = 16 for CPU and `nclones` = 60 for MIC) show the timing of the pure MPI version of the code compiled without OpenMP.

to the pure MPI run that is mainly explained by the reduction of memory contention due to the use of a shared memory programming language such as OpenMP. For 8 and 16 clones, `setrho_1` and `setrho_3` have similar timings. This is not surprising as they are both based on a reduction approach. Indeed, using the OpenMP standard, we tried to mimic the OpenMP reduction in `setrho_1`. However, `setrho_3` is slower for `nclones` $< 8$. We think that this is because, in our case, the problem is too simple and thus dominated by the OpenMP overhead.

For the sorted particles without vectorization, we first observe that `setrho_2` is greatly improved because, as the particles are now sorted, the OpenMP `atomic` occurrences are reduced. Furthermore, due to the enhanced data locality, the timings are improved for all methods. On the gain graph below, we show the time gain of the optimal method, `setrho_1`, as compared to the unsorted case. The maximum gain of 44% is obtained for the pure OpenMP case and decreases to 10% for the pure MPI run.

Similarly, the third column of the figure shows the timings in the sorted and vectorized case. Note that we have now included `setrho_4` as it requires a full sorting and is vectorized by default. This method has the advantage of not being subject to race conditions and not using additional memory. However, it is only beneficial for high number of threads (`nthread` $> 8$) otherwise, the method is the slowest for `nthread` $< 4$. For the other methods, the conclusions are the same as for the sorted case. The vectorization allows one to have a small gain for a number of clones between one and four but it becomes negative for higher values. It has to be noted that the vectorization is poor because there are too few particles per grid cell. Indeed, as we will see later, the performance of the vectorization depends on the number of particles per grid cell; it needs a high particle density to be efficient.

In summary, for all CPU cases, `setrho_1` with homemade reduction is the fastest method. The method `setrho_3` using OpenMP `reduction` has a similar timing as the first method for `nclones` $\geq 8$ but becomes surprisingly slower for `nclones` $< 8$. Although greatly improved by the sorting, `setrho_2` has a poor performance compared to the reduction methods because of the OpenMP `atomic` directives. Finally, `setrho_4` uses no additional memory and is not subject to race conditions. Furthermore, for `nthread` $= 16$, it is the second fastest candidate but otherwise, it is among the slowest of our charge deposition methods.

For the MIC case, the best option is not as obvious. Indeed, depending on the number of clones, different `setrho` methods are optimal. Even though `setrho_4` is more than two times faster for pure OpenMP, we have chosen, for the following, `setrho_2` (with `atomic` directives) because it is the second fastest around pure OpenMP, where we expect the program to run fastest, and it is not constraint to sorted particles with vectorization. Furthermore, where it is not optimal, only a small difference is observed compared to the fastest method.

The sorting allows one to reduce the timing of `setrho_2` by factors from 1.5 to 3 for the same reasons as the CPU case. Essentially, with sorted particles, the `atomic` occurrences are reduced.

Similarly to the CPU case, the performance gain due to the vectorization is very small because too few particles per grid cell are present. It has been checked, see Figure 6, that with around 10 times more particles per grid cell, i.e. a total of 10 million particles, the vectorization performance is enhanced. In the CPU case, the overall gain is of 40% with an impressive gain of a factor 3.7 for `setrho`. For the MIC, the overall gain with 10 million

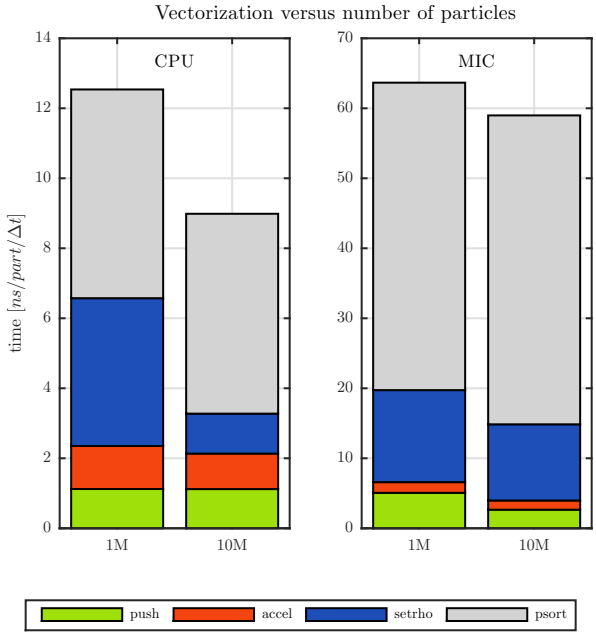


Figure 6: Performance gain due to the vectorization with different number of particles for both CPU (left) and MIC (right). In each case, the timings of `push`, `accel`, `setrho` and `psort` are presented for the standard case with 1 million particles and a test case with 10 million particles.

particles is of $\sim 10\%$. This is less than the CPU case because the MICs have vector registers twice as big as the CPU registers and they need much more particles per grid cell to fully benefit from the vectorization. In our application, we could not test the program with much more than 10 million particles due to the limited memory capacity.

In all cases shown in Figure 5, the unconnected data points at `nclones` $= 16$ for the CPU and `nclones` $= 60$ for the MIC show the timings of the `pic_engine` compiled as pure MPI, i.e. without OpenMP. As expected, the first three methods have the same timings because they rely on the same algorithm, only the OpenMP parallelization is different. Surprisingly, the timings of pure MPI are between 30% and 90% (!) lower than the runs with one OpenMP thread. This is explained by the OpenMP overhead introduced to manage the threads and indicates that it is better to compile the code without OpenMP if we want to use it for a pure MPI run.

The timings of the MICs are on average more than five times higher than CPUs. This is mainly because the MIC heavily relies on vectorization and in our case, there either no vectorization or too few particles per grid cell to benefit from it.

### *3.4. Comparison of the different clusters*

In the previous sections, we have identified the optimal methods for the data structure, the sorting, and charge deposition. We found that the SOA data structure with `psort_2` and `setrho_1` were the most efficient methods on CPU while `psort_2` and `setrho_2` were the most efficient on MIC. In the following, we study the overall timings

of the `pic_engine` on Helios and then compare the results with the other machines, Piz Daint and Piz Dora.

In Figure 7, we show the timings of the different components of the `pic_engine` for the unsorted, and sorted without and with vectorization cases on a single Helios CPU node.

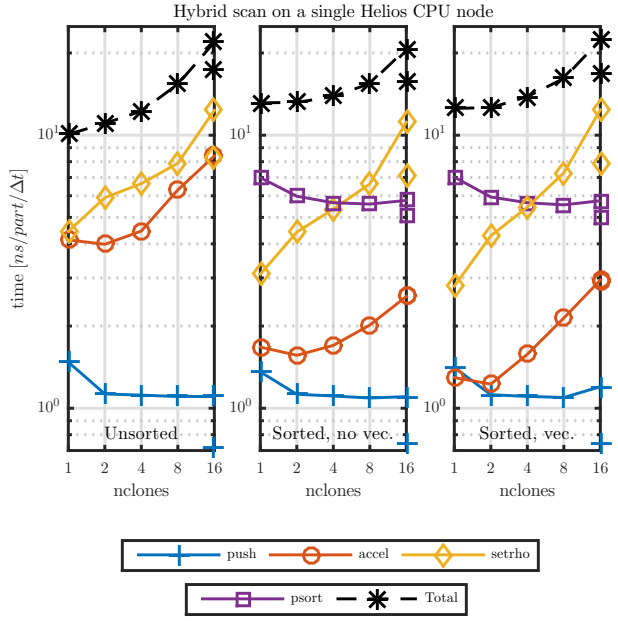


Figure 7: Complete hybrid scan on a single Helios CPU node using the most efficient methods, namely `psort_2` and `setrho_1`. The unconnected points at `nclones` = 16 show the timing of the pure MPI version of the code compiled without OpenMP.

The hybrid implementation using OpenMP allows one to reduce the total timing by a factor of around 2.15 in the unsorted case. Indeed, with a hybrid approach, fewer data replications are made among the clones which improves the memory usage.

When the sorting is activated, the timings of both `accel` and `setrho` are reduced by a factor of 3.25 and 1.44 respectively due to the increased data locality in the memory. However, we note that the total timing is effectively slower than the unsorted case as a result of the sorting cost. For now, the sorting does not lead to a total gain in performance as it represents around 70% of the timing but it is expected that it will turn out to be advantageous for application codes as ORB5 with more complex equations of motion and higher interpolation schemes. Indeed, the cost of the sorting, as previously stated, depends only on the number of particles and buckets whereas the cost of `accel`, `push` and `setrho` will increase as the complexity of the physics/numerics increases.

With vectorization, the timings of both `accel` and `setrho` are further decreased by $\sim 25\%$ and $\sim 8\%$ respectively for `nclones` $\leq 8$ and are increased as we approach the pure MPI runs. This loss of performance is explained by the small particle density per grid cell that makes the vectorized loops too short to be efficient.

The single data points located at `nclones` = 16 represent the timing of the `pic_engine` when compiled without OpenMP. As for the charge deposition, it is seen that OpenMP, even with a unique thread introduces a lot of overhead as the program runs around 30% faster without OpenMP activated.

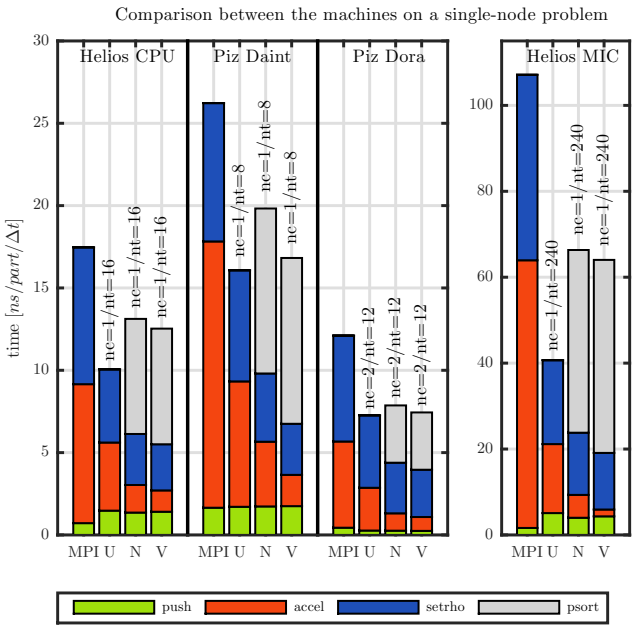


Figure 8: Comparison of the performance on the different computers considered in this study. For each machine, the best timing of the MPI version and the unsorted and sorted with and without vectorization cases are shown. Respectively, they correspond to MPI, U, N and V in the graph. Above each hybrid result, the number of clones and threads used to obtain this timing is shown. Note that for the MIC, hyperthreading has been used with four threads per core.

In Figure 8, a comparison between Helios CPU and MIC nodes, and the CSCS machines Piz Daint and Piz Dora is shown. The timings are obtained as follows. For each case, a hybrid scan similar to Figure 7 is made and the minimum total timing is retained. For each machine, four cases are presented. First, the rough timing of the `pic_engine` made in the pure MPI version of the code compiled without OpenMP (MPI) is shown. Then, the usual unsorted (U) and sorted without (N) and with vectorization (V) timings are presented.

As we will see shortly, the conclusions are the same for all the machines but differ quantitatively as the architectures and compilers are different.

The hybrid unsorted version of the code is very beneficial for the performance with a time gain of at least 60% compared to the pure MPI due to the reduced memory contention. Furthermore, the sorting and vectorization add a significant performance gain on both `accel` and `setrho` on a single node.

Note that except for Piz Dora, the optimum timings are obtained for a pure OpenMP run with one clone. For Piz Dora, the optimal configuration is found for two clones and twelve threads which corresponds to one clone per socket.

Finally, the performance of the MIC is surprisingly poor compared to the CPU timings, $\sim 5$ times slower for our simple application. In fact, we have shown that for an increased particle density per grid cell, the MIC performance is enhanced. Indeed, the main advantage of the MICs is their improved vectorization with 512-bits vector registers but it is bound in our application by the small number of particles per grid cell. Note that for

these results the optimal performance was found for a pure OpenMP run with 4 (!) hardware threads per physical core.

### *3.4.1. Multinode study*

Singlenode runs as presented previously are a useful way to provide insight into the program performance. However, for production runs, we generally need to use several nodes to partition the large problem size and number of particles across the cluster nodes. For this reason, in this section, we present multinode timings of the `pic_engine`. Specifically, we will be able to measure the timings of the inter-node communications that are present in `setrho` to transfer the guard cells between subdomains and in `pmovez` that transfers the particles between the subdomains.

The results presented in this section are made on 16 nodes of the different machines. To be able to compare them with the singlenode results, the cases tested here correspond to 16 copies of the singlenode problem. In other words, we evolve $16 \times 10^6$ particles on a $512 \times 256 \times 16$ grid such that each node has the equivalent of a singlenode problem. Furthermore, the timings are again given in $ns$ per particle and per timestep normalized to one node, i.e. for one million particles.

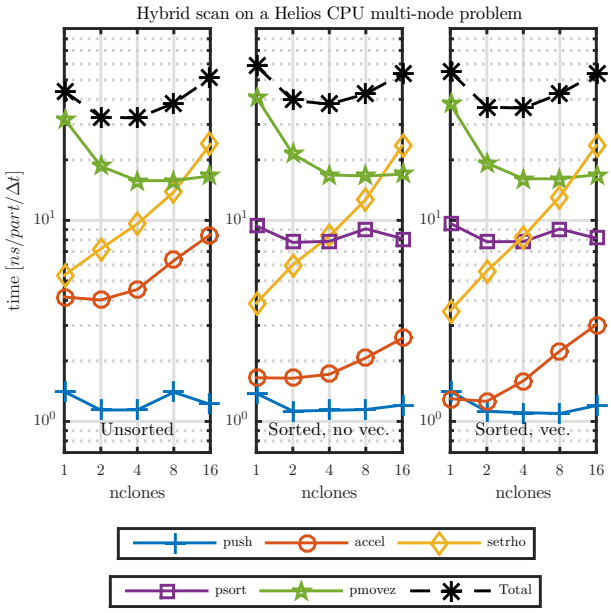


Figure 9: Hybrid scan on 16 Helios CPU nodes using the most efficient methods, namely `psort_2` and `setrho_2`. The single points show the timing of the pure MPI version of the code compiled without OpenMP.

In Figure 9, we show the multinode hybrid scan on the Helios computer for the unsorted, and sorted without and with vectorization cases. The timings are qualitatively very similar to the singlenode runs. The main difference comes from the `pmovez` routine that is now activated and that represents around half of the total timing. In our case, less than 1% of the particles per subdomain are moved. Thus, the communication time is negligible. However, the `pmovez` routine spends a lot of time preparing the communication by, for example,

determining the particles that should be moved which represent, in this case, most of the `pmovez` timing. Similar to the sorting method, this is not an issue for some application codes as with a higher physical complexity, the cost of `pmovez` will not change. Indeed, it has been found in ORB5 that it does not represent more than 10% of the total timing for all production runs made so far.

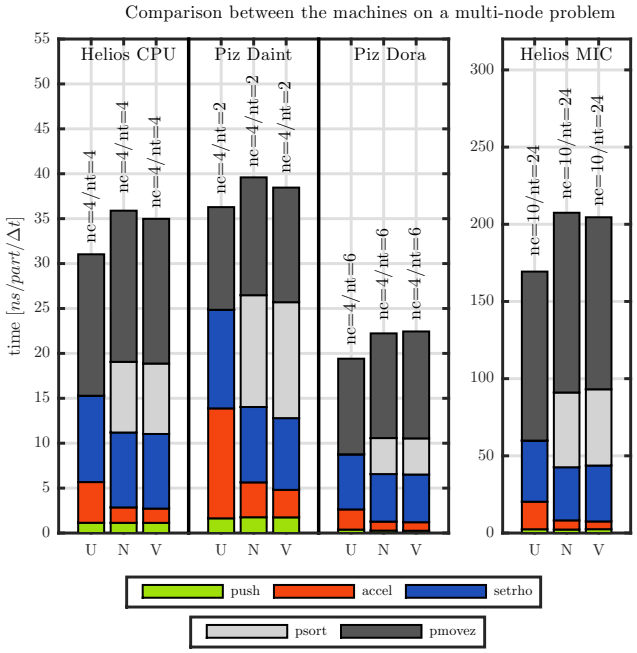


Figure 10: Comparison of the performance of a 16 nodes run on the different computers considered in this study for a multinode simulation. For each machine, the unsorted (U) and sorted without (N) and with (V) vectorization cases are shown. Above each hybrid result, the number of clones and threads used to obtain this timing is shown. Note that for the MIC, hyperthreading has been used with four threads per core.

In Figure 10 are shown the timings of the multinode hybrid runs on all the machines. As for Helios, we see that the `pmovez` method dominates all the timings. This conclusion is even enhanced for the MIC that suffers from a poor communication network as the messages must first pass through the CPU before being sent to the recipient MIC. Again, let us emphasize that only the improvements of `setrho` and `accel` are relevant as they are the most time-consuming methods of PIC codes such as ORB5.

The single and multinode results are compared in Table 3. As expected, the timings of `push` and `accel` do not vary between the runs.

The timings of `setrho` are slightly higher in the multinode case because we have enabled MPI communications between the subdomains to transfer the guard cells. This represents around 20% of the total `setrho` timing.

Surprisingly, the sorting method is slower for the multinode runs. This is however explained by the fact that in multinode simulations, the particles travel through the subdomains in the $z$ direction and their positions are not sorted in the $(x, y)$ plane in the recipient subdomain.

| | Helios CPU | | Piz Daint | |
|---|---|---|---|---|
| **Nodes** | Single | Multi | Single | Multi |
| `push` | 1.41 | 1.37 | 1.75 | 1.76 |
| `accel` | 1.30 | 1.30 | 1.90 | 1.93 |
| `setrho` | 2.80 | 3.56 | 3.10 | 3.54 |
| `psort` | 7.02 | 9.66 | 10.07 | 12.12 |

| | Piz Dora | | Helios MIC | |
|---|---|---|---|---|
| **Nodes** | Single | Multi | Single | Multi |
| `push` | 0.73 | 0.73 | 4.33 | 5.01 |
| `accel` | 0.81 | 0.81 | 1.59 | 1.69 |
| `setrho` | 2.76 | 3.28 | 13.18 | 15.93 |
| `psort` | 4.65 | 5.43 | 44.93 | 47.65 |

Table 3: Comparison of the performance for the single- and multinode simulations. The timings shown here correspond to pure OpenMP runs and are presented in $ns/particle/\Delta t$ normalized to the singlenode problem to compare the results.

## 4. Conclusions

With the goal of porting application PIC codes to the new multi- and manycore HPC platforms, we have developed and tested different optimization techniques and parallelization schemes. For this purpose, we have developed a test bed called the `pic_engine` as a simple and portable abstraction of the PIC algorithm allowing us to easily test these techniques on a simplified yet realistic code. To that end, only the key elements common to all PIC applications are retained.

In this work, we have considered a 6D Vlasov equation describing the evolution of a single-species plasma in the electrostatic limit in Cartesian coordinates. For the sake of simplicity, no magnetic field is considered and no field solver is used. The charge depositions and grid-to-particle interpolations are done using, respectively, a cloud-in-cell method and a linear interpolation on the electrostatic potential. The time integration scheme evolving the equations of motion is based on the second order leap-frog method.

In order to preserve the portability of the program, it has been coded using Fortran and a hybrid implementation of MPI and OpenMP for the parallelization. By doing so, the code can be run on most of the modern HPC clusters including the recent MIC-equipped computers.

We have presented different possible optimizations, which have been tested on a singlenode problem in order to apprehend the shared memory programming model and the inherent difficulties.

We have first discussed the data structure used in the `pic_engine` to store particle quantities. Both structure of arrays and array of structures have been considered. Since most of the memory access is made following the particle order, it is found that the structure of arrays is more efficient because data is accessed contiguously — recall that Fortran uses a column-major order to store arrays in memory. A performance gain up to 14.4% is observed in this mode.

In most of the PIC codes, particles are stored randomly in computer memory. This is critical for the program performance since random memory accesses are very inefficient. To avoid this problem, we have implemented a counting sort algorithm to sort the particles according to their positions. Despite its cost, the sorting allows increasing the performance of the most time-consuming methods, namely `accel` and `setrho`, by respectively 3.25 and 1.44. Furthermore, the particle sorting also allows vectorizing both `accel` and `setrho`, which is otherwise not possible due to their indirect addressing. A performance gain up to 25% is observed with the vectorization for the case considered with an average of eight particles per grid cell. Furthermore, we have shown that with an increased number of particles per grid cell, the vectorization performance increases. For the CPU, the gain is up to 30% while it is around 10% for the MIC. The lower gain increase for the MIC compared to the CPU is explained by the poor vectorization due to the small number of particle per grid cell and the wide 512-bits vector registers used by the MIC that can vectorize up to eight doubles compared to four for the CPU.

The `setrho` method is a challenging candidate for the parallelization because of the race conditions. In the `pic_engine`, we have implemented different parallelization methods based either on an atomic update of the memory or data safe algorithms that completely avoid these race conditions.

The use of a shared memory programming language allows reducing the memory contention problem by limiting data replication thus improving the performance. Indeed, it is observed that the pure OpenMP version of the code runs up to 2.15 times faster than the pure MPI version.

Multinode experiments were done to test the `pic_engine` in more realistic conditions and to time the MPI communications. The total multinode timings are almost twice slower than the singlenode results mainly because of the `pmovez` method used to transfer the particles across the subdomains but they are not representative of more realistic conditions. However, for more complex applications, the timings of `psort` and `pmovez` will remain the same in absolute value, but the timings of the other methods will be much higher due to the more complex physics and/or higher order schemes. Therefore, the relative timings of `psort` and `pmovez` is expected to be much smaller in such applications.

We have been able to test the performance of the `pic_engine` on the new MIC architectures. This is possible because the same programming model is used for both CPUs and MICs in native mode. Unfortunately, the MIC performance is poor compared to the CPU. In this case, we think it is due to the poor vectorization. However, a recently published study [17] shows that with the appropriate low-level optimizations, the Intel MIC can be up to 1.6 times faster than an Intel eight-core CPU. In a future work, we will consider their approach and try to apply it to the `pic_engine`.

Finally, the `pic_engine` is not only a simple abstraction of an application PIC code allowing to design and test easily new optimizations, but, due to its modularity, it can also be used to find the optimal running configuration on any computer.

**Acknowledgments**

The authors would like to thank G. Merlo for all the useful discussions and suggestions. This project has been supported by the Platform for Advanced Computing (PASC) programme. 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. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

## Appendix A. Parallelization algorithms

In this appendix, we present the different parallelizations methods we have implemented in the `pic_engine` for the sorting and the charge deposition already briefly presented respectively in Sections 3.2 and 3.3. The algorithms are presented in Fortran-like language that we have intentionally simplified to keep only the relevant information. They may not work as they are but only the parallelization technique is emphasized here.

### *Appendix A.1. Sorting*

We have seen in Algorithm 1 in Section 3.2 that the counting sort used in the `pic_engine` consists mainly in three loops, two over the particles to create a histogram and put them in their right place and one over the buckets for the prefix sum. Note that in all versions, the prefix sum is never parallelized because it is negligible in our application. Here are the two different methods discussed in this paper:

**`psort_1`:** In this method, the first loop is parallelized using OpenMP `reduction` and the third loop is done with a OpenMP `do` loop and `atomic` directives to avoid race conditions. Furthermore, in the first loop, the `displs` array is calculated as the `count` array of particles in each bucket. It is then overwritten by the *inclusive* scan in the second loop.

```
displs(:)=0
!$Omp Parallel &
!$Omp Private(ind1,ind2,ind)
!$Omp Do Reduction(+:displs)
Do i=1,np
  ! Compute the particle grid-cell
  ! index
  indX=Int(part_att(i,dirX)/dx)+1
  indY=Int(part_att(i,dirY)/dy)+1
  ind=(indY-1)*nbX+indX
  ! Create the histogram
  displs(ind)=displs(ind)+1
End Do
!$Omp End Do

!$Omp Single
! Prefix sum
Do i=2,nbX*nbY
  displs(i)=displs(i)+displs(i-1)
End Do
```

```
!$Omp End Single

!$Omp Do Private(is)
Do i=np,1,-1
  ! Compute the particle grid-cell
  ! index
  indX=Int(part_att(i,dirX)/dx)+1
  indY=Int(part_att(i,dirY)/dy)+1
  ind=(indY-1)*nbX+indX
! Update atomically to avoid race
! conditions
!$Omp Atomic Capture
  is=displs(ind)
  displs(ind)=displs(ind)-1
!$Omp End Atomic
  part_att_temp(is,:)=part_att(i,:)
End Do
!$Omp End Do
!$Omp End Parallel
```

`psort_2`: The last sorting method is slightly different than first. During the first loop, we compute the index of each particle in the sorted array and race conditions are avoided with OpenMP `atomic`. By doing so, the third loop is trivially parallelized with OpenMP `do` loop.

```
!$Omp Parallel &
!$Omp Private(is,ind1,ind2,tmp)
!$Omp Do
Do i=1,np
  ! Compute the particle grid-cell
  ! index
  indX=Int(part_att(i,dirX)/dx)+1
  indY=Int(part_att(i,dirY)/dy)+1
  ind(i)=(indY-1)*nbX+indX
  is=ind(i)
!$Omp Atomic Capture
  tmp=counts(is)
```

```
  ! Create the histogram
  counts(is)=counts(is)+1
!$Omp End Atomic
  indices(i)=tmp
End Do
!$Omp End Do

!$Omp Single
displs(1)=counts(1)
Do i=2,nb1*nb2
  displs(i)=displs(i-1)+counts(i)
End Do
!$Omp End Single

!$Omp Do
Do i=1,np
  is=ind(i)
  part_att_temp(displs(is)- &
     & indices(i),:)=part_att(i,:)
End Do
!$Omp End Do
!$Omp End Parallel
```

*Appendix A.2. Charge assignment*

Similarly to the sorting, we present here different charge assignment methods implemented in the `pic_engine`. The four versions of `setrho` are:

**`setrho_1`:** In this first method, we mimic the OpenMP `reduction` with an in-house algorithm inspired by the OpenMP standard [18].

```
!$Omp Parallel &
!$Omp Private(tid,ix,iy,iz,wx,wy,wz)
! Get the thread number
tid=omp_get_thread_num()

!$Omp Do
```

```
Do ip=1,np
  ! Compute the particle weight
  ix=Int(part_att(ip,1)-xgrid(0))
  wx=part_att(ip,1)-xgrid(ix)
  ! Do similarly for y and z
  ...

  ! Deposit the charge according to
  ! the weights
  rho_loc(ix,iy,iz,tid)= &
     & rho_loc(ix,iy,iz,tid)+ &
     & (1.0-wx)*(1.0-wy)*(1.0-wz)
  ! Do similarly for the other seven
  ! grid points
  ...
End Do
!$Omp End Do

! Perform the reduction
!$Omp Do Collapse(3)
Do iz=0,nz
  Do iy=0,ny
    Do ix=0,nx
      rho(ix,iy,iz)= &
       & Sum(rho_loc(ix,iy,iz,:))
    End Do
  End Do
End Do
!$Omp End Do
!$Omp End Parallel
```

`setrho_2`: The second method uses the OpenMP `atomic` directive to safely update the charge array.

```
!$Omp Parallel &
!$Omp Private(ix,iy,iz,wx,wy,wz,ip)
!$Omp Do
```

```
Do ip=1,np
  ! Compute the particle weight
  ix=Int(part_att(ip,1)-xgrid(0))
  wx=part_att(ip,1)-xgrid(ix)
  ! Do similarly for y and z
  ...

  ! Deposit the charge according to
  ! the weights
!$Omp Atomic Update
  rho(ix,iy,iz)=rho(ix,iy,iz)+ &
     & (1.0-wx)*(1.0-wy)*(1.0-wz)
  ! Do similarly for the other seven
  ! grid points
  ...
End Do
!$Omp End Do
!$Omp End Parallel
```

**setrho_3:** Similarly to the first method, we use this time the OpenMP `reduction` to avoid race conditions when updating the charge array.

```
!$Omp Parallel Private(ix,iy,iz)
!$Omp Do Private(wx,wy,wz,inb,ic) &
!$Omp Reduction(+:rho)
Do ip=1,np
  ! Compute the particle weight
  ix=Int(part_att(ip,1)-xgrid(0))
  wx=part_att(ip,1)-xgrid(ix)
  ! Do similarly for y and z
  ...

  ! Deposit the charge according to
  ! the weights
  rho(ix,iy,iz)=rho(ix,iy,iz)+ &
     & (1.0-wx)*(1.0-wy)*(1.0-wz)
```

```
    ! Do similarly for the other seven
    ! grid points
    ...
  End Do
  !$Omp End Do
  !$Omp End Parallel
```

**setrho_4:** Finally, in this last method we use the finite support of the CIC method. Indeed, here each particle will only contribute to the eight nearest grid points.

```
  !$Omp Do Parallel Collapse(3) &
  !$Omp Private(ix,iy,iz,d,ibin) &
  !$Omp Private(jbin,wx,wy,wz)
  Do iz=0,nz
   Do iy=0,ny
    Do ix=0,nx
     d=0.0
     Do jbin=Max(1,iy),Min(ny,iy+1)
      Do ibin=Max(1,ix),Min(nx,ix+1)
       bin=(jbin-1)*nx+ibin
       Do ip=displs(bin)+1, &
         & displs(bin+1)
         wx=part_att(ip,1)-xgrid(ix)
         wy=part_att(ip,2)-ygrid(iy)
         wz=part_att(ip,3)-zgrid(iz)
         d=d+Min(1.0-wx,1.0+wx)* &
         & Min(1.0-wy,1.0+wy)* &
         & Min(1.0-wz,1.0+wz)
       End Do
      End Do
     End Do
     rho(ix,iy,iz) = d
    End Do
   End Do
  End Do
  !$Omp End Parallel Do
```

## References


[1] F. H. Harlow, A Machine Calculation Method for Hydrodynamic Problems, Los Alamos Scientific Laboratory report LAMS-1956.

[2] C. K. Birdsall, A. B. Langdon, Plasma Physics via Computer Simulations, CRC press, 2004.

[3] R. Hockney, J. W. Eastwood, Computer Simulation Using Particles, CRC press, 1988.

[4] W. Lee, Gyrokinetic particle simulation model, Journal of Computational Physics 72 (1) (1987) 243–269. doi:10.1016/0021-9991(87)90080-5.
URL http://www.sciencedirect.com/science/article/pii/0021999187900805

[5] Z. Lin, Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations, Science 281 (5384) (1998) 1835–1837. doi:10.1126/science.281.5384.1835.
URL http://science.sciencemag.org/content/281/5384/1835.abstract

[6] T. M. Tran, K. Appert, M. Fivaz, G. Jost, J. Vaclavik, L. Villard, Global gyrokinetic simulation of ion-temperature-gradient-driven instabilities using particles, in: Theory of fusion plasmas, Societa Italiana di Fisica, 1999, p. 45.

[7] Y. Idomura, S. Tokuda, Y. Kishimoto, Global gyrokinetic simulation of ion temperature gradient driven turbulence in plasmas using a canonical Maxwellian distribution, Nuclear Fusion 43 (4) (2003) 234–243. doi:10.1088/0029-5515/43/4/303.
URL http://iopscience.iop.org/article/10.1088/0029-5515/43/4/303

[8] F. Hariri, T. Tran, A. Jocksch, E. Lanti, J. Progsch, P. Messmer, S. Brunner, C. Gheller, L. Villard, A portable platform for accelerated PIC codes and its application to GPUs using OpenACC, Computer Physics Communicationsdoi:10.1016/j.cpc.2016.05.008.
URL http://linkinghub.elsevier.com/retrieve/pii/S0010465516301242

[9] V. K. Decyk, T. V. Singh, Adaptable Particle-in-Cell algorithms for graphical processing units, Computer Physics Communications 182 (3) (2011) 641–648. doi:10.1016/j.cpc.2010.11.009.
URL http://www.sciencedirect.com/science/article/pii/S0010465510004558

[10] X. Kong, M. C. Huang, C. Ren, V. K. Decyk, Particle-in-cell simulations with charge-conserving current deposition on graphic processing units, Journal of Computational Physics 230 (4) (2011) 1676–1685. doi:10.1016/j.jcp.2010.11.032.
URL http://www.sciencedirect.com/science/article/pii/S0021999110006479

[11] Intel, Intel Xeon Phi Product Family (2016).
URL http://www.intel.eu/content/www/eu/en/processors/xeon/xeon-phi-detail.html

[12] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T. Tran, B. F. McMillan, O. Sauter, K. Appert, Y. Idomura, L. Villard, A global collisionless PIC code in magnetic coordinates, Computer Physics Communications 177 (5) (2007) 409–425. doi:10.1016/j.cpc.2007.04.006.
URL http://www.sciencedirect.com/science/article/pii/S0010465507002251

[13] R. Teyssier, Cosmological hydrodynamics with adaptive mesh refinement, Astronomy and Astrophysics 385 (1) (2002) 337–364. doi:10.1051/0004-6361:20011817.
URL http://dx.doi.org/10.1051/0004-6361:20011817

[14] N. Ohana, Private Communication (2015).

[15] D. Hilbert, Über die stetige Abbildung einer Linie auf ein Flächenstück,, Mathematische Annalen 38 (1891) 459–460.

[16] A. Jocksch, F. Hariri, T.-M. Tran, S. Brunner, C. Gheller, L. Villard, A Bucket Sort Algorithm for the Particle-In-Cell Method on Manycore Architectures, Springer International Publishing, Cham, 2016, pp. 43–52. doi:10.1007/978-3-319-32149-3_5.
URL http://dx.doi.org/10.1007/978-3-319-32149-3{_}5

[17] I. Surmin, S. Bastrakov, E. Efimenko, A. Gonoskov, A. Korzhimanov, I. Meyerov, Particle-in-Cell laser-plasma simulation on Xeon Phi coprocessors, Computer Physics Communications 202 (2016) 204–210. doi:10.1016/j.cpc.2016.02.004.
URL http://www.sciencedirect.com/science/article/pii/S0010465516300194

[18] OpenMP Specifications (2015).
URL http://openmp.org/wp/openmp-specifications/