Proceedings Paper:
Lantra, Zaman, Wright, Steven A. orcid.org/0000-0001-7133-8533 and Mudalige, Gihan R. (2024) OP-PIC - An Unstructured-Mesh Particle-in-Cell DSL for Developing Nuclear Fusion Simulations. In: 53rd International Conference on Parallel Processing Gotland, Sweden. The 53rd International Conference on Parallel Processing, 12-15 Aug 2024 ACM , SWE

https://doi.org/10.1145/3673038.3673130

OP-PIC – An Unstructured-Mesh Particle-in-Cell DSL for Developing Nuclear Fusion Simulations

Zaman Lantra
zaman.lantra@warwick.ac.uk
University of Warwick
United Kingdom

Steven A. Wright
steven.wright@york.ac.uk
University of York
United Kingdom

Gihan R. Mudalige
g.mudalige@warwick.ac.uk
University of Warwick
United Kingdom

ABSTRACT

Particle-in-Cell (PIC) applications form a core simulation component for designing fusion reactors and their efficient use. In this work, we introduce OP-PIC, a new embedded Domain Specific Language (DSL) for developing unstructured-mesh PIC applications. The DSL is aimed at gaining performance portability for PIC codes on current and emerging, massively parallel architectures. We investigate and bring together the state-of-the-art in PIC solver parallelization techniques, refactoring them within a multi-layered DSL. OP-PIC use source-to-source translation to generate platform-specific optimizations. These parallelizations can be reused for any application declared using the DSL's high-level API. We showcase the performance and portability of two non-trivial PIC applications developed with OP-PIC on multiple CPU and GPU clusters, employing a number of parallelization techniques, including OpenMP, CUDA, HIP and their combinations with distributed memory (MPI) parallelization. We benchmark the OP-PIC generated code on a range of single node systems and a number of distributed-memory systems, including an AMD EPYC CPU-based HPE-Cray EX cluster, an NVIDIA V100 GPU cluster, and an AMD MI250X GPU cluster, exploring both single node and scaling performance. Results demonstrate the flexibility of the DSL to implement radically different optimizations for each platform, showing between 1.4× to 3.5× speed-ups with GPUs compared to CPUs on power equivalent systems and good weak-scaling to over 10 billion particles.

CCS CONCEPTS

• **Computing methodologies** → *Parallel computing methodologies*;
• **Software and its engineering** → **Domain specific languages**.

KEYWORDS

Particle-In-Cell, PIC, DSL, OP-PIC, Unstructured-mesh

ACM Reference Format:
Zaman Lantra, Steven A. Wright, and Gihan R. Mudalige. 2024. OP-PIC – An Unstructured-Mesh Particle-in-Cell DSL for Developing Nuclear Fusion Simulations. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673130

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673130

1 INTRODUCTION

The particle-in-cell (PIC) method is a well-established numerical technique, used to model the behavior of charged particles under the influence of electrostatic and/or electromagnetic fields [7]. PIC simulations are particularly common in plasma physics to study the behavior of charged particles in plasma environments, notably the research on energy generation through nuclear fusion reactions.

The fundamental PIC algorithm involves tracking particles in a Lagrangian frame, while modeling field values on a stationary Eulerian mesh. Traditionally, the domain is represented as a structured mesh, where the connectivity is implicit, enabling easy identification of adjacent mesh elements. The regular and ordered manner of the data lends well to compiler optimizations, but may not be well suited to modeling highly complex geometries, and simultaneously obtain accuracy and performance [4]. Unstructured meshes, on the other hand, are highly flexible and can conform to complex geometries. Additionally mesh refinement and adaptations are easier to implement, allowing for higher resolutions in regions of interest. However, the irregular data structures, including indirect accesses involving explicit neighbor mappings and non-uniform access patterns, may degrade performance if not carefully implemented.

PIC simulations are typically memory-bound due to low computational intensity in the particle algorithms. There are substantial numbers of particles involved in such simulations, ranging from several millions to trillions [5]. This gives rise to memory bottlenecks when loading large numbers of particles to for low-compute intensity calculations. The issues compound further on modern massively parallel architectures, where feeding large numbers of processor units with work via relatively limited communication channels significantly curtails achievable performance. A further challenge lies in the diversity of such systems that then require platform-specific optimizations, such as optimizing for memory bottlenecks, that lead to multiple divergent code paths, implemented to gain the best performance from them. For many large-scale scientific applications, maintaining multiple versions of a code-base, often in tens of thousands or even millions of lines of code, is infeasible given the significant time and effort required, not to mention the expertise needed in the many different associated technologies. This challenge of achieving performance portability – gaining best performance on multiple architectures/systems without significant manual modifications to the code – ultimately reduces the continued scientific delivery from the applications. Performance portability, together with productivity, a measure of how fast an application can be augmented or extended with new features while maintaining existing code, makes a triumvirate of challenges that "are now general to any scientific domain that relies on numerical simulation software using HPC systems" [38].

A number of PIC applications [4, 5, 41] have achieved a level of performance portability using C++ template libraries such as Kokkos [15] and Alpaka [24], or using similar `ParallelFor` targets for CUDA, HIP, and DPC++ [35]. In this work we use an alternative technique, based on creating higher-level abstractions for the class of applications, and optimized automatic code-generation to develop a custom Domain Specific Language (DSL) for PIC operations. We are specifically motivated by on-going work to develop new modeling software to treat the complex dynamics of high temperature fusion plasma for the design of a nuclear fusion reactor at UKAEA through the NEPTUNE (NEutrals and Plasma TUrbulence Numerics for the Exascale) project [2, 33]. In this paper, we explore the flexibility afforded by a DSL in gaining performance portability, following techniques developed in other domains such as CFD [27, 37], seismological modeling [22], finite element [29], and finite difference [23] solver/application development. More specifically, we make the following contributions:

- We introduce OP-PIC, a new DSL consisting of a high-level API for declaring both electrostatic and electromagnetic PIC on unstructured meshes. OP-PIC follows the loop-level abstraction of OP2 [27] for unstructured meshes, but allows a developer to declare particles and a mesh. It then allows to elucidate the main steps of a PIC application consisting of the key communication/-computation pattern of particle moves and their charges affecting the mesh as an iterative explicit solver;
- OP-PIC brings together the state-of-the-art and best known optimizations for parallelizing PIC on multi-core and many-core processors. The framework uses source-to-source translation to automatically generate optimized versions for multi-core CPUs and GPUs (OpenMP, CUDA, HIP, and their combinations with MPI). This includes multiple distributed memory algorithms, optimizations for double indirections for particle moves, and platform specific optimizations for GPUs, such as segmented reductions;
- Finally, we benchmark the performance of two non-trivial PIC applications, Mini-FEM-PIC [39] and CabanaPIC [25], developed with OP-PIC on CPU and GPU clusters. Results are presented on an Intel CPU cluster, a large AMD CPU cluster (ARCHER2), a NVIDIA GPU cluster, and a large AMD GPU cluster (LUMI-G). Both single-node and weak scaling performance for a number of representative problem sizes are provided. We present an analysis of performance bottlenecks and discuss lessons learnt in the development of the DSL.

We show that the high-level abstraction techniques used in OP-PIC can provide a clean separation of concerns between the declaration of the solver (the science source) and its parallel implementation. A developer can use the OP-PIC API to write a solver as a serial implementation without worrying about data races, synchronizations, or explicit data copies between host and device; OP-PIC handles the parallelization *orchestration* automatically. In contrast, a Kokkos developer for example needs to address data races, possibly by using scatter views, and handle synchronizations through barriers/fences. Additionally, unlike the `ParallelFor` methods, users of OP-PIC will not need to develop their own distributed memory parallel implementation. A further advantage is the greater flexibility provided by the source-to-source translation techniques for generating optimized target code compared to other methods, significantly easing the implementation (and reuse) of radically different parallelizations. We see matching or marginally better performance from OP-PIC generated code, compared to the Kokkos version of CabanaPIC. With the OP-PIC generated code, we see approximately 1.4× to 3.5× speed-up gained with GPUs compared to CPUs on power equivalent systems, and good weak scaling up to 1024 GPUs.

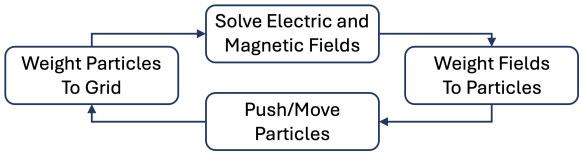


Figure 1: Core PIC iterative algorithm

The remainder of this paper is structured as follows: Section 2 introduces the general PIC algorithm, the underlying formulation, and related work; Section 3 outlines the proposed DSL including its API, backend design, and automatic code-generation targeting multiple parallelizations; Section 4 details performance results on a range of parallel systems; finally, Section 5 concludes the paper.

2 THE PARTICLE-IN-CELL (PIC) METHOD

In a PIC simulation, charged particles move over a simulation domain under the influence of electric and magnetic fields. The domain is represented by an Eulerian mesh, whereas the particle movement is tracked using Lagrangian methods. Particle-to-particle interactions are typically not directly computed, instead particles influence the electric and magnetic fields, that in turn affect the movement of particles. This gives rise to an *operator split* approach as described at a high-level by Brown et al. [9]. Here we use this description to elucidate the core algorithm in order to understand the key computation and communication pattern of PIC for the purposes of developing a DSL. In general the algorithm consists of four main steps, as summarized in Figure 1: (1) compute the electric and/or magnetic fields (Field solver), (2) compute the effect of the electric and/or magnetic fields on each individual particle (Weight fields to particles), (3) compute the movement of particles due to fields, and move particles (Push/Move particles), and (4) compute the effect of updated particle positions onto the mesh (Weight particles to grid).
1. Field solver: The algorithm begins by computing the electric/-magnetic fields using Maxwell's equations which govern time evolution of electric and magnetic fields:

$$\nabla \cdot \vec{E} = \frac{\rho}{\epsilon_0} \quad (1) \qquad \nabla \cdot \vec{B} = 0 \quad (2)$$

$$\frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E} \quad (3) \qquad \frac{\partial \vec{E}}{\partial t} = \frac{1}{\mu_0 \epsilon_0} \nabla \times \vec{B} - \frac{1}{\epsilon_0} \vec{J} \quad (4)$$

The equations consist of Gauss' Law (1), the magnetic divergence constraint (2), Faraday's Law (3), and Ampere's Law (4). Here, $\vec{E}$ is the electric field, $\vec{B}$ is the magnetic field, ρ is the charge density, $\vec{J}$ is the current density, ϵ_0 is the vacuum permittivity, and μ_0 is the vacuum permeability. When the electric field is irrotational (i.e., $\nabla \times \vec{E} = 0$), the scenario is regarded as electrostatic, leading to a reduction of the equations to only Gauss' Law (1). However, in cases where particles move at relativistic velocities or when the current density is substantial, the complete set of Maxwell's equations must be calculated, classifying the case as electromagnetic [4].

For a structured mesh, the above equations are typically solved using a finite-difference time-domain (FDTD) method (as in VPIC [5] and CabanaPIC [25]). For an unstructured mesh, the finite element (FE) method is typically used [4]. Thus, the field values required for the solution are stored as data structures of the mesh, which can be within cells, nodes and/or edges/faces of the mesh.

2. Weighting fields to particles: The updated electric and magnetic field values are only known at spatial grid points, e.g., mesh cells, but their influence on each particle within the cell needs to be determined. This requires the exact location of each particle. For structured meshes, this process can involve a straightforward interpolation based on distance, however, for unstructured mesh the computation is more involved. The electric and magnetic fields at a specific particle position $\vec{x}_i$ within a cell can be computed using Eq. (5) and Eq. (6):

$$\vec{E}(\vec{x}_i) = \sum_{j=0}^{n_{\text{edge}}} E_j \hat{e}_j(\vec{x}_i) \quad (5) \qquad \vec{B}(\vec{x}_i) = \sum_{j=0}^{n_{\text{face}}} B_j \hat{b}_j(\vec{x}_i) \quad (6)$$

Here, n_{edge} and n_{face} are the number of edges and faces of the mesh cell (i.e., the finite element containing the particle). E_j and B_j are the field values of the edges and faces of the cell respectively. These field values are weighted based on the particle location $\vec{x}_i$ using basis functions $\hat{e}_j$, the Nédélec edge elements [28], and $\hat{b}_j$, the Raviart-Thomas [8] face elements. Since field values are calculated per particle position, the values can be stored in particle data structures, unless the calculations are merged into the move routine; hence, this routine may require mesh-to-particle interactions.

3. Push/Move particles: With each particle now having a known electric/magnetic field value influencing it, the solver needs to compute their new velocities and positions to progress the simulation forward. The force encountered by a charged particle is given by the Lorentz force equation (Eq. (7)). In PIC simulations, to accurately compute the relativistic particle motion, the standard relativistic equations of motion (Eqs. (8) and (9)) are employed:

$$\vec{F} = q\left(\vec{E} + \vec{v} \times \vec{B}\right) \quad (7) \qquad \frac{\partial \vec{v}}{\partial t} = \frac{\vec{F}}{m} \quad (8) \qquad \frac{\partial \vec{x}}{\partial t} = \vec{v} \quad (9)$$

$\vec{x}$ represents the particle position, $\vec{v}$ is the particle velocity, q is the charge of the particle, m is the mass of the particle, and $\vec{F}$ is the force on the particle. The above equations can be solved using classical/explicit leap-frog numerical schemes, the Boris integration method [31] being the de facto method with a non-zero magnetic field. Other methods such as Velocity Verlet (zero magnetic field giving second-order accuracy [32]), Vay, Higuera, and Cary pushers [31] can also be used.

Given that individual particle details are required without dependencies, this routine becomes directly parallelizable. The routine relocates particles to new positions based on the fields and time step, presenting the challenge of determining the new cell in which the particle resides when dealing with an underlying unstructured mesh. One solution is to identify the residing cell by using a nearest neighbor search and jumping to the most probable neighbor if the current cell is not the final location-containing cell [14]. This can be implemented as an iterative process starting from the initial location. Another approach is to move directly into a cell closer to the final location and search the neighboring cells [1, 16, 33]; however, this requires maintaining additional mapping data structures. This may not be a viable solution for electromagnetic simulations since these may require deposition of charge/current to each mesh element passed along the path of particle movement. It is a key issue that has important performance trade-offs, as discussed later.

4. Weighting particles/deposit to grid: In this final step, particles need to deposit their charge or current to the mesh elements at their new locations. This is a key step for transferring information between the Lagrangian particle representation and the Eulerian grid representation.

Depending on the design, charge may be deposited to grid nodes and current to grid edges. This "scatter" step is detailed by Moon et al. [26] and leads to Eq. (10) giving the charge contribution to the i^{th} node and Eq. (11) giving current contribution to the i^{th} edge (see Figure 2 of Moon et al. [26]):

$$q_i = Q_p W_i^0(r_p) \quad (10) \qquad i_i = \frac{Q_p}{\Delta t} \int_{r_{p,s}}^{r_{p,f}} W_i^1(r_p)\, dL \quad (11)$$

Barycentric coordinates are used here where in Eq. (10), $W_i^j(r_p)$ is the barycentric coordinate of the particle's final position r_p, and Q_p represents the charge of the p^{th} particle. For computing the current contribution in Eq. (11), the routine considers the p^{th} particle moving along a straight path L from position $r_{p,s}$ to $r_{p,f}$ during a Δt duration. Thus, in a typical electrostatic simulation, only the element corresponding to the final particle location r_p receives the charge deposition, while in electromagnetic simulations, the fields are generally assessed on each cell along the particle's path of movement. As such, this computation may be integrated with the particle push routine (Step 3 above). Moreover, considering the possibility of multiple particles simultaneously occupying the same mesh cell at the same time, parallel execution of this routine will lead to data races, unless carefully designed.

In practical implementations of PIC, to enhance computational efficiency and minimize overhead, physical particles are usually represented as samples of a distribution function in phase space, forming macro-particles [17], also known as super-particles. These macro-particles exhibit properties that encompass a central position, a charge assignment function, a momentum distribution function, and various other physical quantities required for the PIC algorithm. In conjunction with the core PIC procedure described above, in some state-of-the-art PIC implementations, additional routines, including particle collisions [19], ionizations [13] and particle injections [17], may be interleaved.

2.1 Related Work

While the computational characteristics of PIC codes are well understood for structured-mesh PIC applications, unstructured-mesh PIC codes are less common and have only been developed more recently [4, 16, 41]. With the use of unstructured meshes, these codes aim to account for geometric details and/or localized behaviors of interest using strongly graded, anisotropic meshes.

Well known structured-mesh PIC applications include VPIC [5], PSC [18], SMILEI [13], hPIC [20], EPOCH [6], and PIConGPU [10] developed for simulating a range of scientific phenomena such as laser-plasma interactions, high-power lasers interacting with

matter, astrophysical plasmas, the propagation of electromagnetic waves and near-surface Plasma–Material Interaction (PMI) problems. VPIC version 2.0 [5] uses the Kokkos C++ template library framework to achieve performance portability, while PIConGPU uses Alpaka [24] to obtain performance portability on GPUs and CPUs [40]. Warp [34] and Warp-X [17, 35], for electrostatic and electromagnetic simulations respectively, also employ structured meshes but incorporate adaptive mesh refinement (AMR) with the use of the AMRex library which enables targeting clusters of both CPUs and GPUs. These enables capturing high-fidelity geometries for accuracy with extreme resolutions, resulting in significant computational expenses.

Unstructured-mesh PIC implementations include XGC [12, 41], GTC [36], EMPIRE-PIC [4], and NESO/NESO-Particles [16]. EMPIRE-PIC [4] from Sandia National Laboratories resolves electromagnetic wave propagation through plasma using Finite Element Method PIC calculations and uses Kokkos for gaining performance portability. XGC and GTC, both started as structured-mesh implementations, later evolving to use unstructured meshes. GTC specializes in simulating the 5-dimensional Vlasov equations for magnetic confinement fusion in toroidal geometry, while XGC focuses on simulating multi-physics in the edge region of magnetically confined plasmas. GTC is coded in C, employing a hybrid of MPI and OpenMP, with a CUDA version for GPUs. The unstructured version of XGC-m [41] is developed using the PUMIPic library [14], which incorporates MPI and the performance-portable Cabana library, making it capable of running on both CPUs and GPUs. NESO/NESO-Particles [16] is a code designed to manage unstructured-mesh PIC simulations through spectral finite element methods through the Nektar++ library [11]. It is currently under development as part of the NEPTUNE project [33] with an initial implementation in C++ using SYCL for performance portability.

3 OP-PIC

In this work, we are motivated by the challenge of gaining performance portability for PIC solvers. Our approach is to define a high-level abstraction to achieve separation of concerns where the declaration is decoupled from its parallel implementation(s).

3.1 The OP-PIC API

For unstructured-mesh PIC, the key computation/communication patterns are: (1) computations over mesh elements accessing data on neighboring elements, and (2) particle-to-mesh element interactions, including particle movement along the mesh. Unstructured-meshes use explicit connectivity information to specify the mesh topology. Then operations over mesh elements will entail iterating over all the elements (e.g., cells) of the mesh, accessing/updating data on neighboring elements (e.g., nodes) via a mapping (e.g., a cells-to-nodes mapping) that specifies connectivity. This leads to indirect accesses, the main motif of unstructured-mesh computations. The first part of our PIC abstraction therefore, will be in declaring the mesh, its connectivity between elements (mapping), and then particles that interact with the mesh. The second part will handle operations over the mesh and particles, including moving particles.

3.1.1 Mesh, particle and data declarations: To illustrate the API, consider the mesh in Figure 2(a), consisting of 9 cells (C1–C9) and

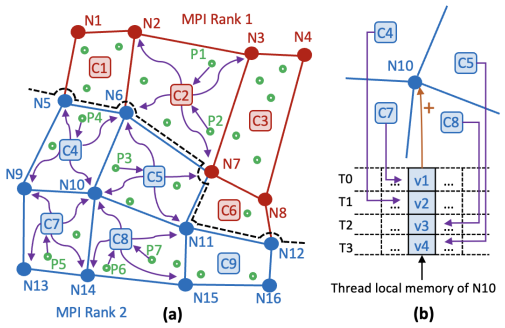


Figure 2: (a) Unstructured mesh with particles, and double indirection in loops over particles, (b) Scatter arrays for handling data races

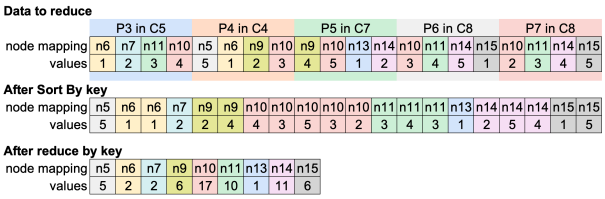


Figure 3: Segmented Reductions

```
1  int nnodes = 16; int ncells = 9;
2  opp_set nodes = opp_decl_set(nnodes, "nodes");
3  opp_set cells = opp_decl_set(ncells, "cells");
4
5  int ngam = 100000;
6  opp_set x   = opp_decl_particle_set("x"  , cells);
7  opp_set gam = opp_decl_particle_set("gam", cells, ngam);
8
9  int* c2n = {1,2,5,6,   2,3,7,6,  3,4,7,8,   5,6,9,10,  ... };
10 int* c2c = {2,4,-1,-1, 1,3,5,-1, 2,6,-1,-1, 1,5,7,-1, ... };
11
12 opp_map cn = opp_decl_map(cells, nodes, 4, c2n, "cell_to_nodes_map");
13 opp_map cc = opp_decl_map(cells, cells, 4, c2c, "cell_to_cell_map" );
14
15 opp_map p2cell_i = opp_decl_map(x,cells, 1, nullptr,"particles_to_cells_index");
16
17 double* d_efield = {...}; double* d_potent = {...}; double* d_cvol = {...};
18 double* d_det    = {...}; double* d_sd     = {...}; double* d_ncd  = {...};
19
20 opp_dat efield = opp_decl_dat(cells, 1, OPP_REAL, d_efield, "electric field");
21 opp_dat sd     = opp_decl_dat(cells, 1, OPP_REAL, d_sd,     "shape deriv");
22 opp_dat cvol   = opp_decl_dat(cells, 1, OPP_REAL, d_cvol,   "cell volume");
23 opp_dat cdet   = opp_decl_dat(cells, 1, OPP_REAL, d_det,    "cell determinants");
24 opp_dat sd     = opp_decl_dat(cells, 1, OPP_REAL, d_sd,     "shape deriv");
25 opp_dat np     = opp_decl_dat(nodes, 2, OPP_REAL, d_potent, "node potential");
26 opp_dat cd     = opp_decl_dat(nodes, 2, OPP_REAL, d_ncd,    "charge density");
27
28 opp_dat pc     = opp_decl_dat(x, 1, OPP_REAL, nullptr, "particle charge");
29 opp_dat lc     = opp_decl_dat(x, 1, OPP_REAL, nullptr, "basis function weights");
30 opp_dat ppos   = opp_decl_dat(x, 1, OPP_REAL, nullptr, "particle position");
```

Figure 4: API for declaring mesh, particles and data

16 nodes (N1–N16). OP-PIC refers to these classes of elements as `sets`, declaring them with `opp_decl_set` (see Figure 4, lines 2–3). Similarly, particle sets are declared, but will be defined on the mesh elements using `opp_decl_particle_set` (lines 6–7).

Mesh connectivity is declared with `opp_decl_map`, which specifies the connectivity between two sets, e.g., cells-to-nodes (line 12) or cells-to-cells (lines 13), together with the arity of the connection, and raw integer data giving the explicit mappings respectively. Given the mesh is fixed, OP-PIC will assume that these mappings will remain static throughout a simulation. `opp_decl_map` can also

be used to declare a mapping between particles and a mesh set (line 15) where the first argument is a particle set followed by a mesh set and arity of 1; a particle is always mapped to just one mesh element. These maps are dynamic since as particles move through the mesh the mapping changes. We can declare it with a `nullptr` if we do not have a particle distribution at initialization, which will then be allocated (integer data) during simulation. However for non-zero particle counts, a valid integer array should be provided.

Data associated with the mesh and particles are declared with `opp_decl_dat` (lines 20–30). Data represents physical quantities such as the electric field, the node potential, or the charge density. `opp_decl_dat` takes the mesh set or particle set on which the data is defined, its dimension, data type, and either a pointer to the raw data or a null pointer in the case of a zero particle count (allocated during the simulation).

3.1.2 Operations Over Sets: All of the numerically intensive operations in a PIC application can be described as computations over sets, accessing data though the mappings either directly or indirectly. A loop over all the mesh elements, such as computing the electric field on the cells based on the node potentials, is declared with the `opp_par_loop` API call (see Figure 5, lines 6–13). The API specifies an elemental function `compute_electric_field_kernel` (as a function pointer, lines 1–4), a string description of the loop, and the iteration set. The elemental function takes a number of arguments, six arguments in this example, and the parallel loop declaration specifies these using the `opp_arg_dat` API. Each argument consists of the `opp_dat` and its access mode, `OPP_READ` (read only), `OPP_INC` (increment), `OPP_WRITE` (write only) or `OPP_RW` (read and write). Additionally, if the `opp_dat` is accessed indirectly, a mapping is also provided (e.g., cells-to-node map, `cn` used here to access node potential `np`) together with the index used in the mapping for each indirect access. Directly accessed `opp_dats` such as `efield` and `sd` do not involve a mapping.

Similar to iterating over mesh elements, `opp_par_loop` can be used to iterate over all particles on the mesh (see Figure 5, lines 20–26). Here iteration set `x` is a particle set accessing data directly on `opp_dat`, `pc`. The particle-to-cell mapping `p2cell_i` is used to indirectly increment `cd`, an `opp_dat` defined on nodes. This double indirection is facilitated by specifying the cells-to-node mapping `c2n` for each of the `opp_arg_dat` arguments. The `OPP_ITERATE_ALL` allows us to iterate on the full number of particles. However, in PIC codes there will be the need to handle particle injections into the domain as a special case. For example, `OPP_ITERATE_INJECTED` will allow iteration of only the newly injected particles for enrichment.

3.1.3 Particle Move: A key step in a PIC solver is the particle move. OP-PIC can declare a particle move as a special loop over particles as illustrated in Figure 6. If we initially assume an electromagnetic PIC code, then the elemental kernel over all particles will require: (1) specifying computations to be carried out for each mesh element, e.g., cells, along the path of the particle, until its final destination cell; (2) a method to identify if the particle has reached its final mesh cell; (3) computations to be carried out at the final destination mesh cell; (4) actions to be carry out if the particle has moved out of the mesh domain; and, (5) calculate the next most probable cell index to search. A template elemental kernel is illustrated in Figure 6 (lines 1–15). The application developer will need to

```
1  void compute_electric_field_kernel(double* ef, double* sd,
2       double* np0, double* np1, double* np2, double* np3){
3       ...
4  }
5  /* declare parallel loop over mesh elements */
6  opp_par_loop(compute_electric_field_kernel, "Compute Electric Field", cells,
7               OPP_ITERATE_ALL,
8               opp_arg_dat(efield,      OPP_INC ),
9               opp_arg_dat(sd,          OPP_READ),
10              opp_arg_dat(np, 0, cn,   OPP_READ),
11              opp_arg_dat(np, 1, cn,   OPP_READ),
12              opp_arg_dat(np, 2, cn,   OPP_READ),
13              opp_arg_dat(np, 3, cn,   OPP_READ));
14
15 void deposit_charge_on_nodes_kernel(double* x, double* pc,
16      double* cd0, double* cd1, double* cd2, double* cd3){
17      ...
18 }
19 /* declare parallel loop over particles */
20 opp_par_loop(deposit_charge_on_nodes_kernel, "Deposit Charge on Nodes", x,
21              OPP_ITERATE_ALL,
22              opp_arg_dat(pc,                  OP_READ),
23              opp_arg_dat(cd, 0, cn, p2cell_i, OPP_INC),
24              opp_arg_dat(cd, 1, cn, p2cell_i, OPP_INC),
25              opp_arg_dat(cd, 2, cn, p2cell_i, OPP_INC),
26              opp_arg_dat(cd, 3, cn, p2cell_i, OPP_INC));
```

Figure 5: API for loops over mesh elements and particles

```
1  void move_particles_kernel(double* ppos, double* lc, double* cvol, double* cdet){
2       /* computation per mesh elment for particle */
3       ...
4       /* check condition for final destination */
5       ...
6       /* if final destination element - final computation*/
7       OPP_PARTICLE_MOVE_DONE
8       ...
9       /* else if out of domain*/
10      OPP_PARTICLE_NEED_REMOVE
11      ...
12      /* else not final destination element - calculate next cell & move further*/
13      OPP_PARTICLE_NEED_MOVE
14      ...
15 }
16 /* declare parallel move loop over particles */
17 opp_particle_move(move_particles_kernel, "Move Particles", x, cc, p2cell_i,
18                   opp_arg_dat(ppos,           OPP_READ ),
19                   opp_arg_dat(lc,             OPP_WRITE),
20                   opp_arg_dat(cvol, p2cell_i, OPP_READ ),
21                   opp_arg_dat(cdet, p2cell_i, OPP_READ));
```

Figure 6: API for particle move

provide each of the above code segments annotating the relevant blocks with pre-processor statements: `OPP_PARTICLE_MOVE_DONE`, `OPP_PARTICLE_NEED_MOVE`, and `OPP_PARTICLE_NEED_REMOVE`.

These allow the OP-PIC code-generator to insert code to implement the parallel particle move as we will discuss in Section 3.2. The `opp_particle_move`, lines 16–21, will then take this kernel function (and a string literal as its name) together with the particle set, mesh element-to-element mapping (e.g., `cc`), particle-to-mesh element mapping (e.g., `p2cell_i`), and a number of `opp_dats` similar to the `opp_par_loop` API. As can be seen, declaration of the particle move API is more involved than the standard parallel loops over mesh elements of particles. A full example can be see in the OP-PIC codebase. For an electrostatic PIC solver, as discussed in Section 2, the particle move will only affect the final destination mesh cell. In this case only the elemental kernel blocks specifying the method to identify the final destination and computations to be done at the final destination need to be specified. However, the parallel implementation of these cases will differ significantly, as detailed in Section 3.2.

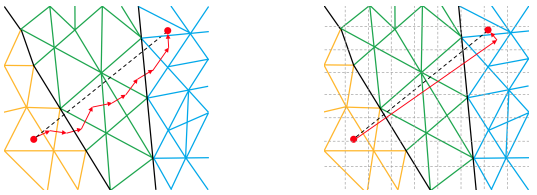
The OP-PIC abstraction was inspired by OP2 [27] and OPS [30], and closely follows the unstructured-mesh declarations in OP2, but extends the API for particle declaration and particle-to-mesh interactions. Similar to OP2, the OP-PIC abstraction assumes that the mesh remains static and there are no mesh refinements during the simulation. However, the implementation of the PIC algorithm on parallel architectures deviates significantly from the parallelizations developed in OP2, for example supporting double indirections for particle-to-mesh contribution scattering and particle movement.

3.2 Parallelizations

OP-PIC implements the operations declared using the API on parallel systems using two main levels: (1) distributed memory and (2) intra-node or shared-memory. These levels operate orthogonally to each other and therefore distributed memory is handled as a classical library implementation, while parallelization on nodes require implementing the loops over sets (mesh and particle sets) via a SIMD, multi-threading, or SIMT model using code-generation.

3.2.1 opp_par_loop: The same distributed memory and on-node parallelization strategies as OP2 can be used for implementing `opp_par_loops` for operations over the unstructured-mesh elements in OP-PIC. Distributed memory entails partitioning and distributing the mesh among disparate memory spaces. The message passing interface (MPI) is used following the standard halo creation and exchange method with an "owner-compute" model, where the partition (i.e., MPI process) which owns the element (such as a cell) will be responsible for performing computations over it. Data races when parallelizing iterations that increments data held on a set, modified indirectly via a mapping, are handled with redundant computations over MPI halos [27]. Within an MPI process, to exploit thread-level parallelization, e.g., targeting a shared-memory multi-processor or a single GPU, the data races are handled using scatter arrays and atomics/segmented reductions, respectively. In scatter arrays (see Figure 2(b)), for each thread a local array is used where, when iterating over mesh cells in parallel, the contribution from a cell to a node (computed by a thread) will be added to the corresponding thread array. Finally, the array entries can be reduced to get the total contribution to that node.

An `opp_par_loop` over particles requires a different parallel implementation. Particles are also distributed over the partitions, but do not contain MPI halos. Particles are related to the mesh through a particle-to-cell map, `p2cell_i`. An MPI process owns the particles related to the cells in that MPI rank, and these particles change as the simulation progresses. This will only occur during particle move or injection. A loop over particles could be direct (e.g., updating data declared on particles), have one level of indirection (e.g., updating data on cells via a particle-to-cell map), or have a double indirection. An example double indirection is the case where a loop over particles deposits charge to the mesh nodes. The state at MPI boundaries can be illustrated as in Figure 2(a). Cell C2 on MPI rank 1 will hold node N6 in its halo, however, N6 is owned by MPI rank 2. A loop over particles will use the particle-to-cell map, `p2cell_i`, and the cell-to-node map, `cn`, to increment data on nodes. For example, when particles P1, P2 in cell C2 increment data on N6 (e.g., deposit charge), then the increments are first written to rank 1's node halos and then at the end of the loop over particles, the updated node halos are communicated to rank 2, which can then update the rank 2 owned N6. Naturally, large numbers of particles incrementing the same node leads to significant data races, which again need to be specially handled for the various on-node/shared memory parallel models. For OP-PIC we have implemented safe/unsafe-atomics and segmented reductions as discussed in Section 3.3.



(a) multi-hop **(b) direct-hop**

Figure 7: Particle move in distributed memory parallelization

3.2.2 opp_particle_move: A particle could move: (1) within the same cell; (2) to a different cell within the same MPI rank; or, (3) to a different cell on a different MPI rank. In the first case, a particle's cell index remains unchanged, whereas a new cell index is assigned to a particle otherwise. The third case requires MPI communications to move particle data to a new MPI rank.

One strategy for moving particles to cells is to loop over each particle and "track" its movement from cell to cell by computing the next probable cell. This entails an inner loop per particle which terminates when the final destination cell is reached, based on the conditions specified in the `opp_particle_move` kernel (see Figure 6). This *multi-hop* (MH) strategy is illustrated in Figure 7(a). If a particle crosses an MPI boundary, it is moved to a halo region of the mesh and marked for communication to the halo of the MPI rank owning the cell to which the particle has moved. Prior to communication, the OP-PIC backend packs the particle data of all marked particles in a buffer, reducing the number of MPI messages. Packing creates holes in the `opp_dats` as particle data is moved out, hence a hole filling routine runs asynchronously during communication, shifting data from the end of the `opp_dats` to fill the holes. Once the particles are received at the destination MPI rank, the data are unpacked to the end of the respective `opp_dats`.

A second, *direct-hop* (DH) strategy, implements a variation of the particle move method inspired by NESO [1, 16] (see Figure 7(b)). Here the particles are moved directly to a cell closer to the final destination, and then switch to *multi-hop* mode to move to the correct final destination. In this case, OP-PIC creates two structured meshes, overlaid over the unstructured mesh: (1) mapping from structured-mesh cell to unstructured-mesh cells (*cell-map*), (2) mapping from structured-mesh cell to MPI rank of which the unstructured-mesh cell belongs to (*rank-map*). OP-PIC use an MPI-RMA-based global move approach to overcome the challenge of identifying the ranks that are trying to communicate and open MPI channels accordingly. Even though, *direct_hop* reduces unnecessary computations and communications significantly, a higher memory footprint is required for bookkeeping. The memory is drastically reduced in MPI-based backends by maintaining only one copy per shared-memory node using MPI-RMA; however, when not using MPI one copy of the cell-map is sufficient.

3.3 Platform Specific Optimizations

A key bottleneck in the solver is the double indirect increment during particle contributions to the mesh elements (e.g., charge deposition from particles to nodes, mapped through cells). However, different hardware requires different data race handling methods to gain best performance. Atomics are generally easier to program and are preferred, but CPUs do not have fast hardware atomics support, hence we use scatter arrays for OpenMP parallelizations on CPUs. Coloring is another option on CPUs, but require particle arrays to be kept sorted, introducing an overhead.

On modern GPUs, fast double-precision atomics are well supported by hardware, especially on NVIDIA GPUs. However, we see performance degradation when using atomics on AMD GPUs, especially when large numbers of particles write to a single memory location, atomics causes serialization. Two alternatives are to use unsafe atomics and segmented reductions. Atomic operations are typically implemented using a compare-and-swap approach. A different implementation using a read-modify-write approach is available on some AMD hardware, but can potentially lead to incorrect answers if not implemented with care.

Alternatively, in a segmented reduction, increments are carried out in three steps: (1) `store_values_and_keys`, (2) `sort_by_key`, and (3) `reduce_by_key`. For instance (see Figure 3), when particles need to deposit charge to the associated four nodes of the residing cell, data hazards are avoided by initially storing the values to reduce alongside their respective node indices. Subsequently, the values are sorted according to the node indices and then reduced according to node indices.

3.4 Automatic Code-Generation

OP-PIC follows the automatic code-generation techniques and technologies in OP2 and OPS for generating on-node/shared-memory parallelizations. The high-level view of this process is illustrated in Figure 8. An application written in OP-PIC will be parsed by the code-generator using `clang` and will build an abstract syntax tree (AST) of the code. The AST facilitates us to easily traverse and collect information on the API calls, for example the arguments of `opp_par_loops`. The API information together with the elemental kernel functions forms an intermediate representation of the application. These are then used to populate or modify code templates for specific parallelizations. The templates are written in Jinja2 and they are updated using Python. These ideas were developed initially by Balogh et al. [3], but we use simpler technologies to encode the boilerplate code for different parallelizations. The system is also easily extensible where a new parallelization, or optimization could be added as a new template which can then be reused. Currently OP-PIC supports the generation of OpenMP, CUDA and HIP parallel code in combination with MPI.

4 RESULTS

We evaluate OP-PIC using two non-trivial PIC applications. The first, *Mini-FEM-PIC*, is a sequential electrostatic 3D unstructured-mesh finite element PIC code written in C++ [39]. It is based on tetrahedral mesh cells, nodes, and faces forming a duct. Faces on one end of the duct are designated as inlet faces and the outer wall is fixed at a higher potential to retain the ions within the duct.

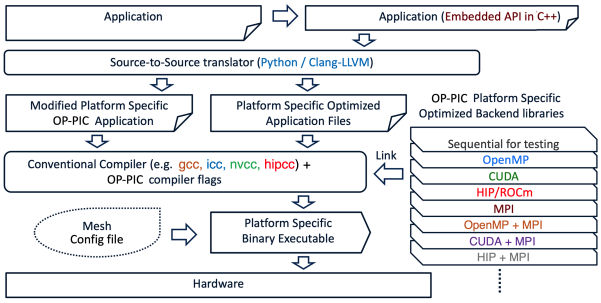


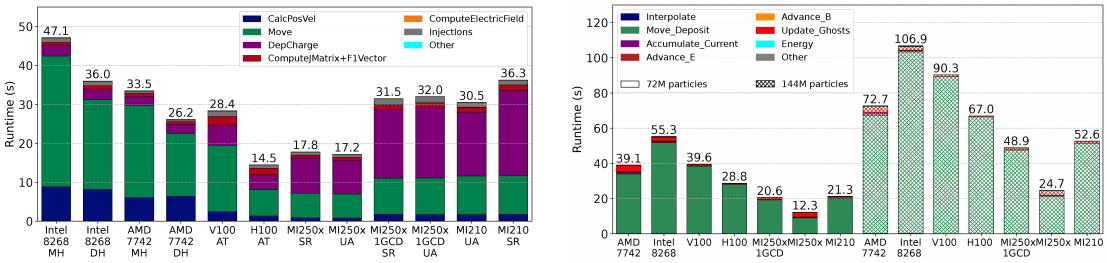
Figure 8: OP-PIC application development life cycle

Charged particles are injected at a constant rate from the inlet faces of the duct (one-stream) at a fixed velocity, and the particles move through the duct under the influence of the electric field. The particles are removed when they leave the boundary face. Overall Mini-FEM-PIC has 1 degree of freedom (DOF) per cell, 2 DOFs per node and 7 DOFs per particle. The second, *CabanaPIC*, is a 3D electromagnetic, two-stream PIC code, where particles move in a duct (cuboid) with cuboid cells. It is implemented with periodic boundaries and has 9 DOFs per cell and 7 DOFs per particle. It was originally developed as part of the Exascale Computing Project, Co-design center for Particle Applications (CoPA) [25] using the Cabana library, based on Kokkos as a structured-mesh PIC application. In the original implementation only shared memory parallelizations are available, but the on-node parallel code can be used within an MPI application if implemented by a user. In this work, we implement the application with OP-PIC, using unstructured-mesh mappings solving the same physics as the original. We validate the electric and magnetic field energy per iteration against results from the original implementation, showing error in the order 10^{-15} (i.e., less than machine precision) in double-precision (FP64).

OP-PIC supports partitioning the mesh with ParMETIS, however, in this paper we use a custom partitioning routine where partitions are created along the "principal direction of motion of particles", as in PUMIPic [14]. This significantly minimizes communication between partitions. Even with a custom partitioning scheme, the developer only needs to provide rank details of a single mesh set. Currently an `opp_dat` can be used to provide this information. OP-PIC will automatically partition the remaining `opp_sets`, communicate to the correct ranks, and create halo regions.

4.1 Single Node/GPU Performance

4.1.1 Runtime and Breakdown: Figure 9(a) presents the runtime and breakdown of time of Mini-FEM-PIC (48k cell mesh with an average of 70M particles) on 2×Intel Xeon 8268 (48 cores) and 2×AMD EPYC 7742 (128 cores) CPUs and a number of GPUs – NVIDIA V100 (32GB), NVIDIA H100 (80GB), AMD MI210 (64GB), and AMD MI250X with 2×Graphics Compute Dies (GCDs) (64GB per GCD). Given Mini-FEM-PIC's electrostatic nature, the electric field remains constant on all particle positions of a cell during the particle move step. This eliminates the need for weighting fields to particle locations where the electric field can be directly inherited from cell data. The push/move routine within Mini-FEM-PIC is divided into two loop functions: `CalcPosVel`, responsible for computing new particle positions and velocities;



(a) Mini-FEM-PIC – 48k cell mesh with 70M particles **(b) CabanaPIC – 96k cell mesh with 144M particles**

Figure 9: Runtime Breakdown on a single node/device

and `Move`, which ensures particles are relocated to the correct cells. The particle-to-grid weighting operation predominantly consists of the `DepositCharge` routine, which may contain data hazards. The field solver encompasses several subroutines, primarily `ComputeJMatrix`, `ComputeF1Vector` and `ComputeElectricField`. The `ComputeJMatrix` and `ComputeF1Vector` routines create the data structures required for a linear solver, which is computed using a PETSc KSP solver.

On CPUs, running Mini-FEM-PIC with flat MPI (which consistently gives better or matching performance compared to OpenMP), the most time-consuming step is the particle move, `Move`; the *direct-hop* (DH) versions gives lower runtimes. On the NVIDIA GPUs, `Move` again dominates runtime. However, on the AMD GPUs, more time is spent in `DepositCharge`. Resolving data races particularly affects performance, where unsafe atomics (UA) gives a marginal improvement over segmented reductions (SR). We observe that standard atomics (AT) on AMD GPUs (not shown here) perform significantly worse, over 200× slower than UA or SR. We attribute this to the serialization of atomics. However, atomics on NVIDIA GPU hardware appear to be better implemented, causing less serialization, so much so that `DepositCharge` is faster than `Move`. Particle sorting is available as an auxiliary API call within OP-PIC; however, periodic shuffling with hole-filling (explained in Section 3.2.2) has proven most effective on GPUs to minimize serialization issues.

The performance of CabanaPIC on the same hardware is detailed in Figure 9(b). In this case we use *multi-hop* (MH) to support the electromagnetic PIC move step. We benchmark two problem sizes with a 96k cell mesh, with 72M (solid filled bars) and 144M particles (cross-pattern bars). The 144M case gives approximately 1500 particles per cell, which is a regime of interest to UKAEA [16]. Instead of directly weighting fields into particle locations, the `Interpolate` routine of CabanaPIC, computes derivatives of field values, storing them as interpolator values within cell data. As the simulation is electromagnetic, each particle must deposit current on all passing cells during the move routine. This entails calculating new positions and velocities, moving to the correct final cell, and depositing current into an accumulator in a single routine, known as `Move_Deposit`. This is the most time consuming routine, overwhelmingly dominating the runtime. Subsequently, the accumulator data is processed in the `AccumulateCurrent` routine to obtain the current density necessary for field solving. Field solving follows a kernel based leap-frog approach, with `AdvanceE` and `AdvanceB` being the key subroutines.

Table 1: GPU utilization

Mini-App	Counts per GCD/GPU	1×MI250X GCDs	8×MI250X GCDs	1×V100 GPUs	4×V100 GPUs
CabanaPIC	96k cell mesh 72M particles	99%	88%	98%	92%
CabanaPIC	96k cell mesh 144M particles	99%	93%	99%	96%
Mini-FEM-PIC	48k cell mesh 70M particles	99%	90%	90%	77%

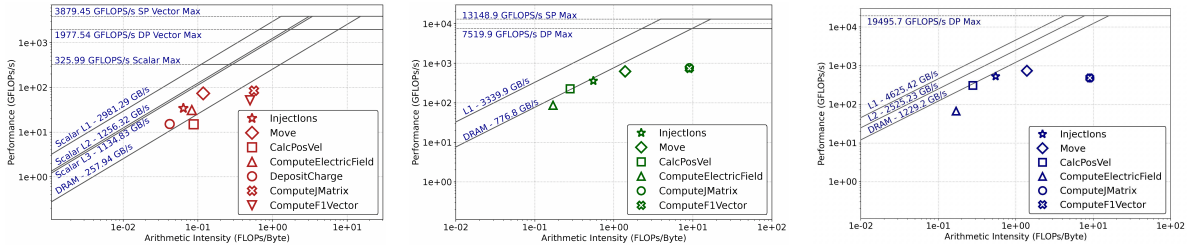
It is interesting to note that in Figure 9(b) the 2×AMD EPYC 7742 gives better performance than the NVIDIA V100, markedly for the 144M particles problem. Here we observe that the GPU suffers from kernel divergence (branching in the kernel) where threads within a warp take different execution paths, leading to inefficient resource use and performance degradation by effectively serializing the execution of threads within the warp. We also see atomics being serialized as an issue especially compounded when there are higher numbers of particles.

Table 1 shows GPU utilization on Bede and LUMI-G (collected with `nvidia-smi` and `rocm-smi`). Both applications show high utilization when using a single GPU, with a noticeable decrease when using multiple GPUs, as MPI communication times and synchronization wait times reduce the GPU utilization. We also see CabanaPIC achieves higher utilization with increased particle counts.

4.1.2 Roofline Performance: Figures 10 and 11 show roofline plots for the significant routines of the OP-PIC generated versions of Mini-FEM-PIC and CabanaPIC for a 2×Intel 8268 CPU node (MPI only), a single V100 GPU, and one GCD of the MI250X GPU. For the Intel 8268 CPU node and the V100 GPU node, we collect data with Intel Advisor and NVIDIA Insight, respectively. For the MI250X GPU, we estimate the arithmetic intensity of kernels and total FP64 operations per kernel using Omniperf[1], and also compare to the FP64 counts for the kernels on the V100 GPU. FLOP/s are then estimated using kernel times from OP-PIC code instrumentation. Finally, the Berkeley ERT [21] is used to obtain rooflines.

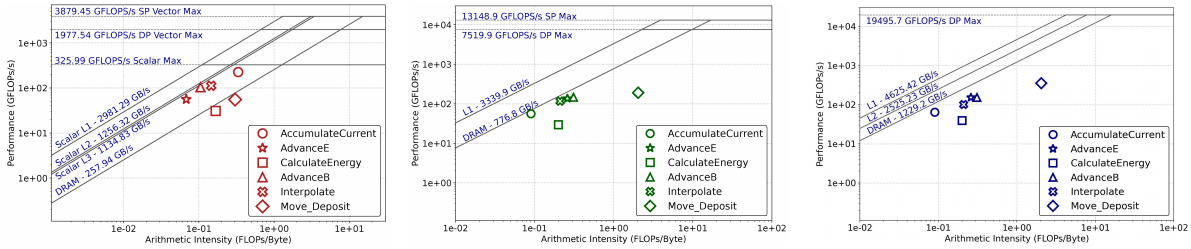
For both applications, on all architectures, almost all routines are bandwidth bound. On CPUs we see several routines L3 cache bound, including the most time consuming `Move` kernel of Mini-FEM-PIC. CabanaPIC's most time consuming CPU kernel, `Move_Deposit`, is just below the threshold of DRAM bandwidth, as it consists of both move and deposit current within the same routine, as opposed to Mini-FEM-PIC's separate routines. Mini-FEM-PIC's

[1] https://rocm.github.io/omniperf



(a) Intel 8268 CPU roofline (b) NVIDIA V100 GPU roofline (c) LUMI MI250X 1GCD roofline

Figure 10: Mini-FEM-PIC single node/device – 48k cell mesh with ≈70M particles



(a) Intel 8268 CPU roofline (b) NVIDIA V100 GPU roofline (c) LUMI MI250X 1GCD roofline

Figure 11: CabanaPIC single node/device – 96k cell mesh with 72M particles

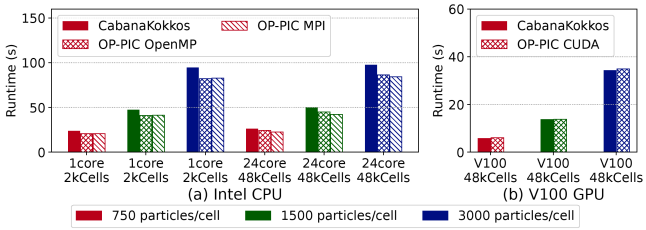


Figure 12: CabanaPIC original vs OP-PIC comparison

`DepositCharge`, does not appear in the GPU rooflines as it is latency bound where thousands of particles are attempting to increment the same mesh element at the same time affecting performance. The `Move_Deposit` routine of CabanaPIC is particularly impacted by kernel divergence, due to significant branching path computations. Additionally, in spite of the notable runtimes of `Update_Ghosts` in Figure 9(b), it is a simple assignment/increment kernel, dominated by halo-exchanges, hence not noted on the roofline plots.

4.1.3 Comparison to State-of-the-Art: Next, we compare the performance of CabanaPIC written using OP-PIC with the original (structured-mesh) version parallelized using Kokkos [25]. We explore three different particle regimes, with 750, 1500, and 3000 particles per cell. We benchmark single-core and single-socket performance on an Intel Xeon 8268 CPU (Figure 12(a)), maintaining constant workload per CPU core per regime, with 2k cells on one core and 48k cells for 24 cores. In all CPU runs, we observe that both OP-PIC OpenMP and MPI outperform the Kokkos version by up to 15%. As noted before, the original code repository does not have a distributed memory parallel version available. Comparing GPU performance using CUDA on an NVIDIA V100 (Figure 12(b)), shows similar performance across all tested regimes. The marginally better

Table 2: Systems specifications

System	Avon Dell PowerEdge C6420	ARCHER2[a] HPE Cray EX	Bede[b] IBM PowerSystems AC922 GPU Cluster	LUMI-G[c] HPE Cray EX GPU Cluster
Processor	Intel Xeon Platinum 8268 Cascade Lake	AMD EPYC 7742	IBM Power9 + NVIDIA Tesla V100-SXM2-32GB	AMD EPYC 7A54 + AMD MI250X
(procs×cores) /node	2×24	2×64	2×16 + 4×GPUs	1×64 + 4×GPUs
Memory/node	192GB	256GB	512GB + 32GB/GPU	512GB + 128GB/GPU
Interconnect	Mellanox ConnectX-6 HDR100 (100 Gb/s) InfiniBand	HPE Cray Slingshot 2×100 Gb/s bi-directional/node	Mellanox EDR (100 Gb/s) Infiniband	HPE Cray Slingshot 50 Gb/s bi-directional/GPU
Compilers	gcc/10.3.0 OpenMPI/4.1.1	PrgEnv-gnu/8.3.3 gcc/11.2.0 cray-mpich/8.1.23	cuda/12.0.1 gcc/12.2 OpenMPI/4.0.5	HIPCC (AMD Clang 14.0.0) Cray Clang 14.0.2
Compiler Flags	`-fopenmp -fPIC -O3`	`-fopenmp -O3`	`-gencode -m64 -O3`	`-isystem -O3`
Petsc Version	PETSc/3.15.1	PETSc/3.20.0	PETSc/3.20.1	PETSc/3.20.0
GPU Arch	-	-	sm_70	gfx90a
Power/node	≈ 475W	660W	≈ 1500W	≈ 2390W

a https://www.archer2.ac.uk
b https://n8cir.org.uk/bede/
c https://lumi-supercomputer.eu/

performance of the OP-PIC version, even though it is implemented using unstructured-mesh computations, is due to the runtime being dominated by the particle move routine `Move_Deposit`, which does not gain any significant advantages on a structured-mesh. Furthermore, the OP-PIC version calculates the next cell using the direction of movement and reading an `int` mapping, whereas the Kokkos version computes the next cell index directly.

4.2 Scaling Performance

We investigate scaling performance for OP-PIC generated code using four cluster systems. The systems are detailed in Table 2, and consists of two CPU clusters (Avon and ARCHER2) and two GPU clusters (Bede and LUMI-G).

We see excellent weak scaling for Mini-FEM-PIC on up to 128 nodes (16k cores) or GPUs (128 MI250X GCDs or 64 V100 GPUs) (Figure 13). In this case, we solve a 48k cell mesh with ≈70M particles per node or GPU for 250 iterations. At each scale, the collection

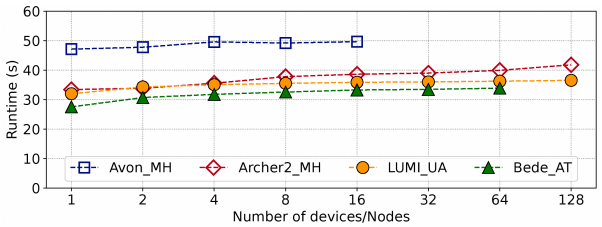


Figure 13: Mini-FEM-PIC weak scaling: 48k cells and 70M particles per CPU node/V100/MI250X GCD for 250 iterations

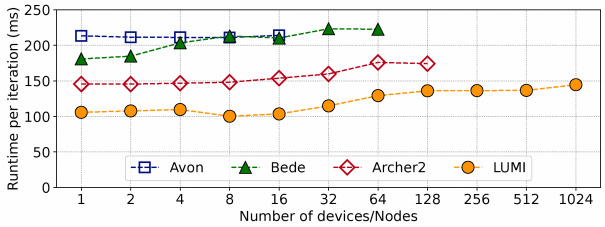


Figure 14: CabanaPIC: 96k cells and 144M particles per CPU node/V100/MI250X GCD

of V100 GPUs or MI250X GCDs perform better than the equivalent number of ARCHER2 nodes. Again, the particle move dominates performance. CabanaPIC also shows good weak scaling – up to 16k cores on ARCHER2 and 1024 GPUs on LUMI-G (see Figure 14). However, solving a 96k cell mesh with 144M particles per node or GPU, the V100 cluster (Bede) performs significantly worse than ARCHER2. This follows from the single-node performance observed in Figure 9(b), where an ARCHER2 node is 20% faster than a single V100 GPU for the same problem configuration per node or GPU. Scaling is also affected by load-balancing of particles, largely determined by mesh partitioning – an equal particle distribution reduces idle time, as finalizing the particle move requires all MPI ranks to synchronize. Comparing MH to DH (not shown) we observed that the DH approach consistently gives 20% faster runtimes.

4.2.1 Power-equivalent Performance: Our final set of results estimate the performance of each machine for a roughly equivalent power envelope. Using the node and GPU power consumption of the systems we estimate that 18 ARCHER2 nodes, 8 Bede nodes (consisting of 32 V100 GPUs) and 5 LUMI-G nodes (consisting of 20 MI250X GPUs) consume roughly 12kW of power.

Running Mini-FEM-PIC with 1.536M cells and ≈2.5B particles, and CabanaPIC with 3.072M cells and ≈2.3B and 4.6B particles, then allows us to compare the speed-ups from the different CPU and GPU systems as illustrated in Figure 15. For Mini-FEM-PIC, the GPU speed-ups compared to ARCHER2 are 1.43× (Bede) and 1.71× (LUMI-G). For CabanaPIC, the speed-ups from LUMI-G are 3.52× and 3.03× for 2.3B and 4.6B particles problems.

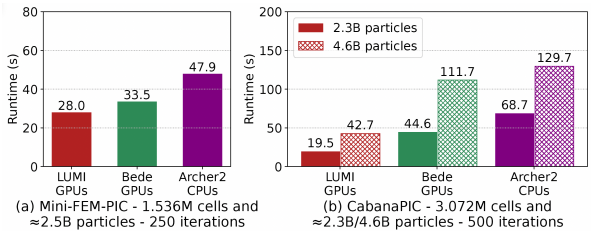


Figure 15: Power equivalent best runtimes – 18 ARCHER2 nodes vs 8 Bede nodes vs 5 LUMI-G nodes

5 CONCLUSION

In this paper, we have introduced OP-PIC, a new embedded DSL for developing unstructured-mesh PIC applications, motivated by nuclear fusion reactor simulations. An application written using the OP-PIC DSL can be used to automatically generate optimized parallel CPU and GPU code. Two non-trivial PIC applications have been developed using OP-PIC and their performance has been explored on a number of single-node and cluster systems. Our results demonstrate the flexibility afforded by automatic code-generation for different code-paths including different optimisation techniques depending on the underlying architecture. Additionally, users of OP-PIC do not need to program the parallelization orchestration, e.g., resolving data races, synchronizations, nor explicit data copies between host and device. Nor do they need to develop their own distributed memory parallelizations. OP-PIC provides optimized MPI-based back-ends, including for example, the much involved particle move step. We see matching or better performance from OP-PIC generated code compared to a state-of-the-art PIC application written using Kokkos, indicating that the high-level abstraction does not detract from gaining good performance. In our experiments runtime is dominated by particle move routines, and almost all steps of the PIC algorithm are bandwidth limited. However, serialization of atomic operations on GPUs leads to a latency-bound charge deposit kernel, which needs further improvements to gain performance. Good weak scaling was observed on 16k CPU cores and up to 1024 GPUs. Comparing performance of CPU and GPU clusters within an equivalent power envelope demonstrates that on the GPU systems we gain 1.4× to 3.5× speed-ups. Future work aims to develop larger and real-world simulations with OP-PIC, implement further optimizations, and extend the code-generation to produce parallelizations for other architectures, such as Intel GPUs. OP-PIC and the mini-apps developed in this paper are available as open source software at: https://github.com/OP-DSL/OP-PIC.

ACKNOWLEDGMENTS

The ExCALIBUR programme (https://excalibur.ac.uk/) is supported by the UKRI Strategic Priorities Fund. The programme is co-delivered by the Met Office and EPSRC in partnership with the Public Sector Research Establishment, the UK Atomic Energy Authority (UKAEA) and UKRI research councils, including NERC, MRC and STFC.

This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk).

This work also made use of the facilities of the N8 Centre of Excellence in Computationally Intensive Research (N8 CIR) provided and funded by the N8 research partnership and EPSRC (Grant No. EP/T022167/1). The Centre is coordinated by the Universities of Durham, Manchester and York.

The authors acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Benchmark Access call (EHPC-BEN-2024B03-043).

Gihan Mudalige acknowledges support from EPSRC (Grant No. EP/V000942/1). The authors thank Will Saunders at UKAEA for his insights on the application domain, and Tobias S. Flynn at Warwick and Istvan Reguly at PPCU Hungary for their advice and support.

REFERENCES

[1] Accessed 2023. NESO Particles Documentation. https://excalibur-neptune.github.io/NESO-Particles/main/sphinx/html/concept/concept.html

[2] Wayne Arter, L. Anton, D. Samaddar, and Rob J. Akers. 2019. *ExCALIBUR Fusion Modelling System Science Plan.* Technical Report CD/EXCALIBUR-FMS/0001. UKAEA. https://www.metoffice.gov.uk/binaries/content/assets/metofficegovuk/pdf/research/spf/ukaea-excalibur-fms-scienceplan.pdf.

[3] G. D. Balogh, Gihan R. Mudalige, István Z. Reguly, S. F. Antao, and Carlo Bertolli. 2018. OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling. In *Proceedings of the IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC).* 59–70. https://doi.org/10.1109/LLVM-HPC.2018.8639205

[4] Matthew T. Bettencourt et al. 2021. EMPIRE-PIC: A Performance Portable Unstructured Particle-in-Cell Code. *Communications in Computational Physics* 30, 4 (Aug. 2021), 1232–1268. https://doi.org/0.4208/cicp.OA-2020-0261

[5] Robert F. Bird et al. 2022. VPIC 2.0: Next Generation Particle-in-Cell Simulations. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (April 2022), 952–963. https://doi.org/10.1109/TPDS.2021.3084795

[6] Robert F. Bird, Patrick Gillies, Michael R. Bareford, Andy Herdman, and Stephen A. Jarvis. 2018. Performance Optimisation of Inertial Confinement Fusion Codes using Mini-applications. *The International Journal of High Performance Computing Applications* 32, 4 (2018), 570–581. https://doi.org/10.1177/1094342016670225

[7] C. K. Birdsall and A. B. Langdon. 1991. Plasma Physics via Computer Simulation. In *Plasma Physics Series.* Institute of Physics Publishing. https://doi.org/10.1201/9781315275048

[8] Franco Brezzi and Michel Fortin. 1991. Mixed and Hybrid Finite Element Method. *Springer Series In Computational Mathematics* 15 (Jan. 1991), 350. https://doi.org/10.1007/978-1-4612-3172-1

[9] Dominic A. S. Brown et al. 2021. Higher-order particle representation for particle-in-cell simulations. *J. Comput. Phys.* 435 (2021), 110255. https://doi.org/10.1016/j.jcp.2021.110255

[10] Heiko Burau et al. 2010. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Transactions on Plasma Science* 38, 10 (2010), 2831–2839. https://doi.org/10.1109/TPS.2010.2064310

[11] Chris D. Cantwell et al. 2015. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications* 192 (2015), 205–219. https://doi.org/10.1016/j.cpc.2015.02.008

[12] Eduardo D'Azevedo et al. 2017. *The fusion code XGC: Enabling Kinetic Study of Multiscale Edge Turbulent Transport in ITER.* CRC Press, 529–552. https://doi.org/10.1201/b21930

[13] J. Derouillat et al. 2018. Smilei: A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications* 222 (2018), 351–373. https://doi.org/10.1016/j.cpc.2017.09.024

[14] Gerrett Diamond, Cameron W. Smith, Chonglin Zhang, Eisung Yoon, and Mark S. Shephard. 2021. PUMIPic: A mesh-based approach to unstructured mesh Particle-In-Cell on GPUs. *J. Parallel and Distrib. Comput.* 157 (2021), 1–12. https://doi.org/10.1016/j.jpdc.2021.06.004

[15] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Extreme Scaling Workshop (XSW'13).* 18–24. https://doi.org/10.1109/XSW.2013.7

[16] ExCALIBUR-NEPTUNE. [n. d.]. NESO Framework. https://github.com/ExCALIBUR-NEPTUNE/NESO and https://github.com/ExCALIBUR-NEPTUNE/NESO-Particles.

[17] Luca Fedeli et al. 2022. Pushing the Frontier in the Design of Laser-Based Electron Accelerators with Groundbreaking Mesh-Refined Particle-in-Cell Simulations on Exascale-Class Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'22)* (Dallas, Texas). IEEE Press, Article 3, 12 pages.

[18] Kai Germaschewski et al. 2016. The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing. *J. Comput. Phys.* 318 (2016), 305–326. https://doi.org/10.1016/j.jcp.2016.05.013

[19] Zoltan Juhasz et al. 2021. Efficient GPU implementation of the Particle-in-Cell/Monte-Carlo collisions method for 1D simulation of low-pressure capacitively coupled plasmas. *Computer Physics Communications* 263 (2021), 107913. https://doi.org/10.1016/j.cpc.2021.107913

[20] Rinat Khaziev and Davide Curreli. 2018. hPIC: A scalable electrostatic Particle-in-Cell for Plasma Material Interactions. *Computer Physics Communications* 229 (2018), 87–98. https://doi.org/10.1016/j.cpc.2018.03.028

[21] Yu Jung Lo et al. 2015. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, 129–148.

[22] M. Louboutin et al. 2019. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development* 12, 3 (2019), 1165–1187. https://doi.org/10.5194/gmd-12-1165-2019

[23] David J. Lusher, Satya P. Jammy, and Neil D. Sandham. 2018. Shock-wave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI. *Computers & Fluids* 173 (2018), 17–21. https://doi.org/10.1016/j.compfluid.2018.03.081

[24] Alexander Matthes et al. 2017. Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the Alpaka Library. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, 496–514.

[25] Susan M. Mniszewski, James Belak, et al. 2021. Enabling particle applications for exascale computing platforms. *The International Journal of High Performance Computing Applications* 35, 6 (2021), 572–597. https://doi.org/10.1177/10943420211022829

[26] Haksu Moon, Fernando L. Teixeira, and Yuri A. Omelchenko. 2015. Exact charge-conserving scatter–gather algorithm for particle-in-cell simulations on unstructured grids: A geometric perspective. *Computer Physics Communications* 194 (2015), 43–53. https://doi.org/10.1016/j.cpc.2015.04.014

[27] Gihan R. Mudalige, Mike B. Giles, István Z. Reguly, Carlo Bertolli, and Paul H. J. Kelly. 2012. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *2012 Innovative Parallel Computing (InPar).* IEEE, 1–12.

[28] Jean-Claude Nédélec. 1980. Mixed finite elements in R3. *Numer. Math.* 35 (1980), 315–341.

[29] Florian Rathgeber et al. 2016. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Software* 43, 3, Article 24 (Dec. 2016), 27 pages. https://doi.org/10.1145/2998441

[30] István Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran, and Simon McIntosh-Smith. 2014. The OPS domain specific abstraction for multi-block structured grid computations. In *The 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing.* IEEE, 58–67.

[31] Bart Ripperda et al. 2017. A Comprehensive Comparison of Relativistic Particle Integrators. *The Astrophysical Journal Supplement Series* 235 (10 2017). https://doi.org/10.3847/1538-4365/aab114

[32] Will Saunders, James Cook, and Wayne Arter. 2022. *ExCALIBUR 1-D and 2-D Particle Models.* Technical Report CD/EXCALIBUR-FMS/0070. UKAEA. https://excalibur-neptune.github.io/Documents/UKAEA_CD-EXCALIBUR-FMS0070-1.00-M4c.1_DDParticleModelsm4C100.html.

[33] Edward J. Threlfall, Rob J. Akers, Wayne Arter, et al. 2023. Software for Fusion Reactor Design: ExCALIBUR Project NEPTUNE : Towards Exascale Plasma Edge Simulations. In *Proceedings of the 29th IAEA Fusion Energy Conference.* GBR.

[34] Jean-Luc Vay et al. 2004. Application of adaptive mesh refinement to particle-in-cell simulations of plasmas and beams. *Physics of Plasmas* 11 (April 2004), 2928–2934. https://doi.org/10.1063/1.1689669

[35] Jean-Luc Vay et al. 2018. Warp-X: A new exascale computing platform for beam–plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 909 (2018), 476–479. https://doi.org/10.1016/j.nima.2018.01.035

[36] Bei Wang et al. 2019. Modern Gyrokinetic Particle-In-Cell Simulation of Fusion Plasmas on Top Supercomputers. *The International Journal of High Performance Computing Applications* 33, 1 (2019), 169–188. https://doi.org/10.1177/1094342017712059

[37] F. D. Witherden, A. M. Farrington, and P. E. Vincent. 2014. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications* 185, 11 (2014), 3028–3040. https://doi.org/10.1016/j.cpc.2014.07.011

[38] Steven A. Wright et al. 2024. Developing performance portable plasma edge simulations: A survey. *Computer Physics Communications* 298 (2024), 109123. https://doi.org/10.1016/j.cpc.2024.109123

[39] Steven A. Wright, Edward Higgins, Gihan R. Mudalige, Ben McMillan, and Tom Goffrey. 2023. *Progress on Development of an FEM-PIC Miniapp.* Technical Report 2057699-TN-03-03. UKAEA. https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/2057699/TN-03-3.pdf.

[40] Erik Zenker et al. 2016. Performance-Portable Many-Core Plasma Simulations: Porting PIConGPU to OpenPower and Beyond. In *High Performance Computing*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.). Springer International Publishing, 293–301.

[41] Chonglin Zhang, Gerrett Diamond, Cameron W. Smith, and Mark S. Shephard. 2023. Development of an unstructured mesh gyrokinetic particle-in-cell code for exascale fusion plasma simulations on GPUs. *Computer Physics Communications* 291 (2023), 108824. https://doi.org/10.1016/j.cpc.2023.108824

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.12793460

ARTIFACT IDENTIFICATION

OP-PIC is a high-level embedded domain specific language (DSL) for writing unstructured-mesh Particle-in-Cell algorithms with automatic parellelization on multi-core and many-core architectures. The API is embedded in C/C++. The current OP-PIC DSL supports generating code targeting multi-core CPUs with OpenMP threading, many-core GPUs with CUDA/HIP offloading, and distributed memory cluster variants of these using MPI.

This appendix contains the information necessary to compile and run the OP-PIC DSL library and the applications described in this paper. It includes: (1) the code/scripts, meshes, and configuration files that were used in the experiments; and, (2) a Dockerfile for CPU deployment on a single computer. A detailed step-by-step deployment guide, together with the necessary environment variables, libraries, and compilers required to deploy OP-PIC on both CPU and GPU architectures can be found in the comments at the end of the Dockerfile.

This paper presents experiments with two PIC applications, Mini-FEM-PIC and CabanaPIC, developed using the DSL. We describe the system environments, setup, and workflow in support of reproducing the results from these applications in this artifact.

REPRODUCIBILITY OF EXPERIMENTS

Example Slurm submission scripts for the main four systems, Avon, ARCHER2, Bede, and LUMI-G, are included in the artifact archive showing the specific commands used (see the `script_files` directory) . See the README file of the artifact for more details on the configurations used for the experiments. All figures are generated through Python matplotlib files which can be found in the `python_diagram_files` directory.

(1) Single node runtimes (**Paper Section 4.1.1**) can be obtained by running the command manually on the terminal, or by submitting the Slurm scripts (in `script_files` directory) with the number of nodes/GPUs set to one. Each simulation typically takes 1-3 minutes to run. Mini-FEM-PIC single node runs use a 48,000 cell mesh with $1e^{18}$ plasma density, generating ≈70 million particles for the simulation. CabanaPIC experiments use $nx = 40$, $ny = 40$, $nz = 60$ creating 96,000 cells and 750/1500 particles per cells. The timing data of the solver routines (in the output log) can then be used to recover the single node breakdown behaviour described in this paper. More details and an example of an output log can be seen in the README file of the artifact.

(2) Roofline experiments (**Paper Section 4.1.2**) are done using the NVIDIA Nsight Compute for the V100 GPU, and with Intel Advisor for the Intel CPU. For the MI250X GPU, we estimate the arithmetic intensity of kernels and total FP64 operations per kernel using Omniperf, and also compare to the FP64 counts for the kernels on the V100 GPU. FLOP/s are then estimated using kernel times from OP-PIC code instrumentation. The rooflines (bandwidth and peak FLOP values) were obtained from Intel Advisor for the Intel CPU plot, and the GPU systems were benchmarked with Lawrence Berkeley National Laboratory's Empirical Roofline Tool (ERT).

(3) The original CabanaPIC comparisons (**Paper Section 4.1.3**) use the master branch of the CabanaPIC repository: https://github.com/ECP-copa/CabanaPIC/ with b7dc525daa110146 commit. The installation requires Kokkos and the Cabana library. This is compared against the MPI, OpenMP, and CUDA versions of OP-PIC on an Intel Cascade Lake CPU socket and on an NVIDIA V100 GPU. A number of minor adjustments were made to the original CabanaPIC code which are described in the README file of the archived artifacts. The runs were done manually and take 20-30 minutes to execute.

(4) Weak scaling runtimes (**Paper Section 4.2**) are collected using the Slurm scripts provided in the `script_files` directory. This directory is sub-divided into `application_name` and then into `server_name`. Since particles per cell is controlled by configurations, the mesh size is increased according to the number of CPU nodes or V100 GPUs or MI250X GCDs used. This increases both the mesh size as well as the particle count with the expected amount for weak scaling. Meshes with multiples of 48k cells are used for Mini-FEM-PIC (48k, 48k×2, 48k×4, 48k×8, ...) and meshes with multiples of 96k cells are used for CabanaPIC (96k, 96k×2, 96k×4, 96k×8, ...). The `MainLoop TotalTime` from the log files are collected to plot the weak scaling results. Each script runs the application multiple times and a single application run on one server configuration takes roughly 1-3 minutes for Mini-FEM-PIC and roughly 2-4 minutes for CabanaPIC.

(5) Power-equivalent performance runs (**Paper Section 4.2.1**) are carried out using Slurm scripts and can be found in the artifact archive along with the weak scaling scripts. It is configured to run on 18 ARCHER2 nodes, 5 LUMI-G nodes (with 20 MI250X GPUs), and on 8 Bede nodes (with 32 V100 GPUs). The `MainLoop TotalTime` from the log files are used to plot the results. The total time to obtain power-equivalent runtimes is ≈20 minutes. Mini-FEM-PIC iterates 250 times with a 1.536 million cell mesh and ≈2.5 billion particles. CabanaPIC uses $nx = 40$, $ny = 40$, $nz = 1920$, generating 3.072 million mesh cells and 750/1500 particles per cell, adding up to 2.3/4.6 billion particles, for 500 iterations.

ARTIFACT DEPENDENCIES AND REQUIREMENTS

1. Checklist (artifact meta information)

- **Program:** Mini-FEM-PIC has been tested with PETSc versions 3.15.1 to 3.20.1, but may work on later versions also. There are no third-party library dependencies for CabanaPIC.
- **Compilation:** See **Paper Table 1**. Some example source files can be found in the `source_files` directory of the artifacts, and the Dockerfile contains further details. Makefiles are included to compile the library and applications.

- **Binary:** Binaries for CUDA, HIP, Sequential, OpenMP, MPI, MPI+CUDA, and MPI+HIP can be generated.
- **Data set:** Mesh files used for the scaling and roofline studies. Mini-FEM-PIC can run using HDF5 mesh files (if HDF5 version of the apps are compiled) or ASCII `.dat` mesh files.
- **Runtime environment:** Details of module environments used on Avon, ARCHER2, Bede, and LUMI-G are available in the GitHub repository as well as in the artifact archive.
- **Hardware:** See **Paper Table 1** for details on systems used in this work, namely Avon, ARCHER2, Bede, and LUMI-G clusters. OP-PIC is tested on Intel/AMD CPUs and on NVIDIA/AMD GPUs.
- **Execution:** Example Slurm submission scripts for the clusters are included in the artifacts.
- **Output:** Timing for different routines of the simulation.
- **Experiment workflow:** Install the OP-PIC library and its dependencies (PETSc/HDF5 required for Mini-FEM-PIC). Compile Mini-FEM-PIC and/or CabanaPIC applications, run and observe performance.
- **Experiment customization:** The simulation can be customized using a configuration file, an example file is present in the directory of each application and the cluster specific files are included in the artifact archive.
- **Publicly available?:** Yes.

2. How software can be obtained

- The OP-PIC GitHub repository: https://github.com/OP-DSL/OP-PIC
- The archived artifacts contain a copy of the code used for this paper.

3. Hardware dependencies:

The code has been tested on AMD/Intel CPU based HPC clusters, NVIDIA GPU HPC clusters, and AMD GPU HPC clusters. It can also be compiled for single server nodes or consumer hardware with or without MPI. To reproduce the exact results from the paper, access to the aforementioned systems is required. However, the details from this artifact description provide a complete view of the steps required to run the experiments on similar hardware.

4. Software dependencies

See the top-level README file listing the dependencies, along with where they can be found. The Avon, ARCHER2, Bede, and LUMI-G systems provide a module environment that contain several of these dependencies. The exact module environment used on each machine can be found in the `source_files` in the artifact.

A Python (>= 3.8) installation is required for the code-generator to operate and details on setting-up the code-generation environment (one time process) can also be found in the README file of OP-PIC. Mini-FEM-PIC depends on the HDF5 and PETSc libraries.

5. Datasets

The mesh files used for Mini-FEM-PIC scaling and the Roofline studies can be found in the `mesh_files` directory of the artifact. CabanaPIC generates the mesh during runtime using configurations, hence does not require reading in a mesh from file.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

1. Installation

Once the dependencies are satisfied, activate the Python environment by:

```
$ source $OPP/opp_translator/opp_venv/bin/activate
```

Now the installation of the software can be achieved with the following steps. Check the Dockerfile for further detailed step-by-step information.

a) Build the OP-PIC library

The library can be compiled using the Makefile located in the directory `OP-PIC/opp_lib`. Mini-FEM-PIC requires the OP-PIC library to be linked with PETSc, while for CabanaPIC, the library should be built without PETSc. See the README file in `OP-PIC/opp_lib` or the Dockerfile for more information.

b) Build Application – Mini-FEM-PIC

The application files can be found in the `OP-PIC/app_fempic` directory. Mini-FEM-PIC can be built with or without HDF5, using `fempic.cpp` for non-HDF5 and `fempic_hdf5.cpp` for HDF5 builds.

(1) **Generate-Code:**
```
$ python3 $OPP_TRANSLATOR -v -I$OPP_PATH/include/
  --file_paths <app_cpp_file>
```
This generates a `fempic_opp.cpp` or `fempic_hdf5_opp.cpp` file and `seq`, `omp`, `mpi`, `cuda`, and `hip` directories.

(2) **Compile the application:** Compile the application with the required parallelization using the Makefile provided in the application directory.

The README file in the `OP-PIC/app_fempic` directory contains more information on these steps. The `OP-PIC/app_fempic_cg` directory contains code-generated files, that can be compiled directly using the make commands, skipping Step (1).

c) Build Application – CabanaPIC

CabanaPIC compilation follows same steps as Mini-FEM-PIC, however it does not contain an HDF5 version. The application files can be found in the `OP-PIC/app_cabanapic` directory.

(1) **Generate-Code:**
```
$ python3 $OPP_TRANSLATOR -v -I$OPP_PATH/include/
  --file_paths cabana.cpp
```
This generates a `cabana_opp.cpp` file and `seq`, `omp`, `mpi`, `cuda`, and `hip` directories.

(2) **Compile the application:** Compile the application with the required parallelization using the Makefile provided in the application directory.

The README file in the `OP-PIC/app_cabanapic` directory contains more information on these steps. In addition, the directory `OP-PIC/app_cabanapic_cg` contains code-generated files, that can be compiled directly using the make commands, skipping Step (1).

2. Running the applications

Once the required parallelized version of the OP-PIC application is compiled, a configuration file and a mesh file (mesh file only for Mini-FEM-PIC) are required for execution. Compiled binaries can be found in a `bin` directory within the application directory.
`$ <app_binary> <config_file>` can be used to run non-MPI simulations and `mpirun` can be used for MPI parallelized applications.