T/AW088/22
Software Support Procurement

Report 2067270-TN-03-01

D2.1 – Progress Towards DSL Adoption
Revision 1.0

Steven Wright, Edward Higgins, and Christopher Ridgers

University of York

Gihan Mudalige and Zaman Lantra

University of Warwick

May 19, 2023

Changelog

1 Summary

Project NEPTUNE (**NE**utrals & **P**lasma **TU**rbulence **N**umerics for the **E**xascale) is concerned with the development a new code for the simulation of a next generation fusion reactor. Both fluid and particle models will be required by such a complex simulation code, along with methods of coupling the two models. In NEPTUNE, the fluid model is likely to take the form of a high-order finite element method, while the particle model will necessarily be particle-in-cell (PIC).

In our previous report (2057699-TN-04-01) we proposed a domain specific language (DSL) for particle-in-cell simulations, providing the API calls specific to these simulations. The DSL that we propose right now is a loop-level abstraction, specialising code generation for the hardware. A higher-level abstraction allowing developers to specify problems in terms of mathematical equations can be built on top of this, targeting the DSL as the backend.

This report documents our progress towards developing a DSL that can be used for PIC methods and presents performance results for an application developed using the proposed DSL.

1.1 The Particle-in-Cell Method

The PIC method is a well established procedure for modelling the behaviour of charged particles in the presence of electric and magnetic fields [1, 2]. Discrete particles are tracked in a Lagrangian frame, while the electric and magnetic fields are stored on stationary points on a fixed Eulerian mesh.

The electric and magnetic fields evolve according to Maxwell's equations (Equations (1)-(4)).

$$\nabla \cdot \vec{E} = \frac{\rho}{\epsilon_0} \tag{1}$$

$$\nabla \cdot \vec{B} = 0 \tag{2}$$

$$\frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E} \tag{3}$$

$$\frac{\partial \vec{E}}{\partial t} = \frac{1}{\mu_0 \epsilon_0} \nabla \times \vec{B} - \frac{1}{\epsilon_0} \vec{J} \tag{4}$$

While the force experienced by a particle is calculated according to the Lorentz force (Equation (5)).

$$\vec{F} = q\left(\vec{E} + \vec{v} \times \vec{B}\right) \tag{5}$$

A typical PIC method can be thought of as two coupled solvers where one is responsible for updating the electric and magnetic fields according the Maxwell's equations, while another calculates the movement of particles according to the Lorentz force. These are referred to as the *field solver* and the *particle mover* (sometimes called the *particle pusher*), respectively.

The main time loop of the core PIC algorithm consists of: solving the field values on the computational mesh; weighting these values to determine the fields at particle locations; updating the particle velocities and positions; and depositing the particle charge/current back to grid points. The algorithm is summarised in Figure 1.

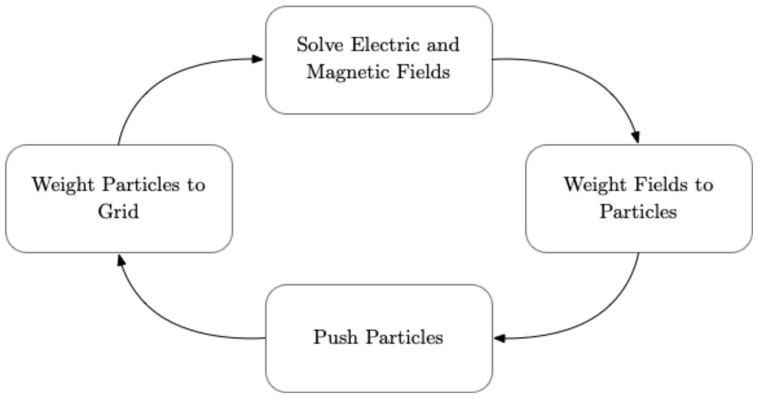
Solve Electric and Magnetic Fields
Weight Fields to Particles
Push Particles
Weight Particles to Grid

Figure 1: Flow chart summarising the key components of the PIC algorithm

Since the field solve acts upon a grid or a mesh, it can easily be implemented using numerous DSLs that have been developed for such simulations (for example, OP2 or OPS). The goal of this work is to develop a DSL extension that allows us to implement the particle mover within the same framework.

2 Developing a PIC Domain Specific Language

Domain Specific Languages (DSLs) allow us to bridge the gap between domain scientists and application developers by allowing the domain specialists to write their calculations using high level abstractions specific to their domain. These abstractions typically take the form an API (Application Programming Interface) embedded in a host language such as C/C++ or Fortran.

A DSL and its associated parser(s)/compiler(s) can then translate this high level abstraction into various low-level parallelisations such as OpenMP, MPI, CUDA, HIP, etc., introducing optimisations to the code using compiler techniques such as source-to-source code translation and code generation. The lower-level implementation focuses on how the computation can be executed in the most efficient way on the given hardware platform, extracting and analysing the computation, data access/communication and synchronisation.

The DSL that we propose right now is a loop-level abstraction, specialising code generation for the hardware. A higher-level abstraction allowing developers to specify problems in terms of mathematical equations can be built on top of this. Similar high-level abstractions have been developed with OpenSBLI[1] (generating OPS loop-level DSL [3]) and Firedrake[2] (generating PyOP2 [4]). Such a higher level abstraction could be developed targeting the OP-PIC DSL as the backend, however we have not focused on this at this time. We will need a better understanding of specifying the problem at a mathematical level for this.

We have identified numerous DSLs for developing structured and unstructured mesh computations (e.g. OP2 [5, 6], OPS [7, 3], Bout++ [8, 9], PATUS [10], UFL [11], PSyclone [12], etc.), but none that include support for particle-in-cell methods. In this report, we detail our progress towards developing a new DSL with a focus on implementing PIC methods. We focus on OP2's loop-level abstraction as a first step towards a proposal for a high-level DSL, like that found in Firedrake [11].

2.1 OP2: A DSL for Unstructured Mesh Computations

OP2 [5] is a high-level abstraction and active library targeting parallel execution of Unstructured mesh applications. It has the capability of auto-generating code for OpenMP, MPI, CUDA, OpenACC and OpenCL, using source-to-source translation. It has a well defined API and the execution algorithm can be divided in to four distinct parts:

(1) Defining sets

(2) Defining connectivity (or mapping) between the sets

(3) Defining data on sets

(4) Operations over sets

[1]`https://opensbli.github.io/`
[2]`https://www.firedrakeproject.org`

For example, a set could be of cells, nodes, edges and/or faces of the mesh; data on sets could be the current over an edge; connectivity could be the mapping between an edge to its connected two nodes; and the operations could be the kernel calculations (solving partial differential equations) by iterating over edges.

Unstructured mesh applications inherently have indirect data accesses, and the main challenges in developing an application will be on data locality, data layout in memory, data dependencies and data conflicts. OP2 handles some of these issues by colouring of the mesh, using atomics (hardware dependent) and partitioning with halo regions. Since the PIC DSL that is to be developed during this research is unstructured mesh, the new development could be inspired by OP2.

2.2 OP-PIC: Unstructured Mesh Particle-in-Cell DSL Design

As stated in Section 1.1, the main loop of a standard PIC algorithm involves four key steps:

(1) Solve Electric and Magnetic Fields (Field Solver)

(2) Weight fields to particles

(3) Push/Move particles

(4) Weight particles to mesh

In many codes, additional routines may also be interleaved, for example, injecting particles or computing particle collisions. In all of these routines the computations typically involves iterating over particles or mesh points (i.e., cells, nodes, edges etc.) and solves mathematical equations such as partial differential equations.

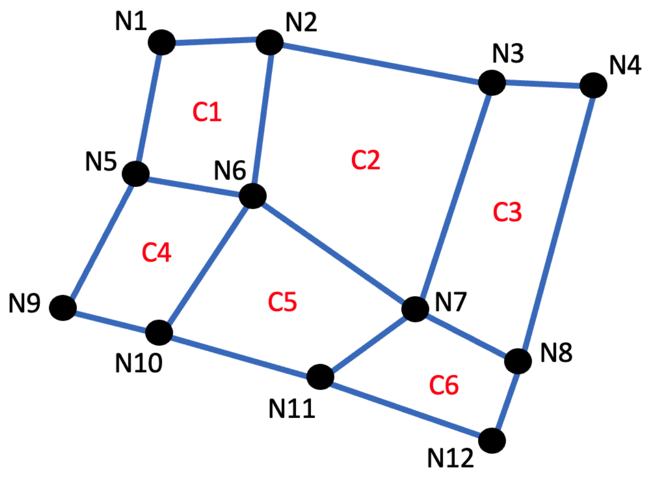

Figure 2: An example unstructured mesh with cells and nodes

Similar to the OP2 execution algorithm (briefly described in Section 2.1), the proposed DSL comprises of the same four distinct parts. Here we give an overview of the API for particle movement within a simple 2D quadrilateral unstructured mesh.

2.2.1 Defining Sets

The mesh in Figure 2 can be defined as a collection of cells (quadrilaterals) and nodes. There are 6 cells and 12 nodes, which can be declared using opp_decl_set.

```
int n_nodes = 12; int n_cells = 6;

opp_set nodes_set = opp_decl_set(n_nodes, "mesh_nodes");
opp_set cells_set = opp_decl_set(n_cells, "mesh_cells");
```

The particle sets can be declared with opp_decl_particle_set allowing multiple particle sets to be defined if there are more than one particle species.

```
opp_set particles_set = opp_decl_particle_set("x particles", cells_set);
```

The above will create an empty particle set, assuming that particles will be injected during the main loop. However, if the initial particle size is known, it could be set when defining the particle set with the API call below.

```
opp_set opp_decl_particle_set(int size, char const *name, opp_set cells_set);
```

2.2.2 Defining Connectivity (or Mapping) Between the Sets

The connectivity is declared through mappings between sets, using opp_decl_map. Considering the mesh in Figure 2, there could be cell_to_node mappings as well as cell_to_cell mappings.

```
int NODES_PER_CELL = 4; int NEIGHBOUR_CELLS = 4;

int* cell_to_nodes = {1,2,5,6, 2,3,7,6, 3,4,7,8, 5,6,9,10, 6,7,10,11, 7,8,11,12};
int* cell_to_cells = {2,4,-1,-1, 1,3,5,-1, 2,6,-1,-1, 1,5,-1,-1, 2,4,6,-1, 3,5,-1,-1};

opp_map cell_to_nodes_map = opp_decl_map(cells_set, nodes_set, NODES_PER_CELL,
                                        cell_to_nodes, "cell_to_nodes_map");

opp_map cell_to_cells_map = opp_decl_map(cells_set, cells_set, NEIGHBOUR_CELLS,
                                        cell_to_cells, "cell_to_cell_map");
```

Each cell belonging to cells_set is mapped to 4 nodes (NODES_PER_CELL) in nodes_set. Hence, the map declaration cell_to_nodes_map has a dimension of 4, thus its indices 0-3 relates to the first cell (C1) mapping its connected { N1, N2, N5, N6 } nodes, indices 4-7 relates to second cell (C2) mapping its connected { N2, N3, N7, N6 } nodes and so on. As shown in int* cell_to_cells, we define -1 as a mapping indicating that there is no element on that direction.

Moreover, since the mapping between particles and cells is dynamic (particles can be injected/removed and they move between cells), we will keep cell_index mapping per particle as data (only the static mesh mappings will be in opp_map).

2.2.3 Defining Data on Sets

Once the sets and its connectivities are defined, the mesh data can be associated with `cells_set` and `nodes_set` through the `opp_decl_dat` API call. Note that in the below example, `node_dat1` is declared with dimension 2, allowing to it store { X, Y } coordinates, while `cell_dat1` stores a single double-precision value per set element.

```
int DIM = 2;
double* d_cell1 = {cd1, cd2, cd3, cd4, cd5, cd6};
double* d_node1 = {x1,y1, x2,y2, x3,y3, x4,y4, x5,y5, x6,y6, x7,y7, x8,y8, x9,y9,
                    x10,y10, x11,y11, x12,y12};

opp_dat cell_dat1 = opp_decl_dat(cells_set, 1, OPP_REAL, (char*)d_cell1, "cell field");

opp_dat node_dat1 = opp_decl_dat(nodes_set, DIM, OPP_REAL, (char*)d_node1, "node field");
```

The particle dats should be created with `opp_decl_particle_dat` and the arguments will be similar to `opp_decl_dat`, except when defining the `cell_index` dat. Here an additional argument "`true`" should be provided indicating that it will be the `cell_index` used to map the particle to its containing cell.

```
opp_dat part_dat1 = opp_decl_particle_dat(particles_set, 1, OPP_REAL, nullptr,
                                                    "part field");

opp_dat part_cell_index = opp_decl_particle_dat(particles_set, 1, OPP_INT, nullptr,
                                                    "part cell index", true);
```

The above will create an empty particle dat, assuming that particles will be injected during the main loop. However, if the initial particle size is known and if the set is created by providing it, the corresponding data could be provided as an array (instead of `nullptr`) using the same API call. This will be helpful when an initial particle distribution is known prior start of the simulation.

```
opp_dat opp_decl_particle_dat(opp_set set, int dim, opp_data_type dtype, char *data,
                               char const *name, bool cell_index = false);
```

2.2.4 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 (if indirection exists).

API: `opp_par_loop`

```
template <typename... T, typename... OPARG>
void opp_par_loop(void (*kernel)(T *...), char const *name, opp_set set,
                    opp_iterate_type iter_type, OPARG... arguments);
```

Consider the following sequential loop, that demonstrates direct and indirect mappings. This uses all of the structures and declarations defined in Sections 2.2.1, 2.2.2 and 2.2.3; however it assumes there are particles in particles_set.

```c
void example_seq_loop(int nparticles, int* cell_to_node, int* cell_idx,
                      double* cell_dat, double* node_dat, double* part_dat) {
    for (int i = 0; i < nparticles; i++) {
        int cell_index = cell_idx[i];

        int node0_mapping = cell_to_node[NODES_PER_CELL * cell_index + 0];
        int node1_mapping = cell_to_node[NODES_PER_CELL * cell_index + 1];
        int node2_mapping = cell_to_node[NODES_PER_CELL * cell_index + 2];
        int node3_mapping = cell_to_node[NODES_PER_CELL * cell_index + 3];

        double inc_value = (part_dat[i] + cell_dat[cell_index]);

        // Assume only X value of node data need to increment
        node_dat[DIM * node0_mapping + 0] += inc_value;
        node_dat[DIM * node1_mapping + 0] += inc_value;
        node_dat[DIM * node2_mapping + 0] += inc_value;
        node_dat[DIM * node3_mapping + 0] += inc_value;

        part_dat[i] = 0.0;
     }
}
```

The sequential example loop above iterates over all the particles, computes the sum of the particle dat and its corresponding cell dat, and increments all 4 connected node dats (only X values) with the sum calculated. Finally the particle dat is assigned with a new value (0.0). Here the particle should map to its containing cell though its cell_index and maps all four nodes connected to that cell, to compute the reduction operation (SUM).

Even though the sequential code looks simple, it would be quite complex if the computation is to be done in parallel (OpenMP, MPI and/or GPUs), due to race conditions (to be considered when executing increments to shared nodes) and data dependencies. However, together with the below API calls and code-to-code translation, the proposed DSL removes all of the development complexities from the domain specialist and will provide an optimised code to run on their intended platform.

```c
void example_kernel(double* part_data, const double* cell_data, double* node0_data,
                    double* node1_data, double* node2_data, double* node3_data) {
    double inc_value = (*part_data + *cell_data);

    // Assume only X value of node data need to increment
    node0_data[0] += inc_value;
    node1_data[0] += inc_value;
    node2_data[0] += inc_value;
    node3_data[0] += inc_value;

    *part_data = 0.0;
}

opp_par_loop(example_kernel, "example_op_par_loop",
    particles_set, OPP_ITERATE_ALL,
    opp_arg_dat(part_dat1, OPP_RW),
    opp_arg_dat(cell_dat1, OPP_READ, OPP_Map_from_Mesh_Rel),
    opp_arg_dat(node_dat1, 0, cell_to_nodes_map, OPP_INC, OPP_Map_from_Mesh_Rel),
    opp_arg_dat(node_dat1, 1, cell_to_nodes_map, OPP_INC, OPP_Map_from_Mesh_Rel),
    opp_arg_dat(node_dat1, 2, cell_to_nodes_map, OPP_INC, OPP_Map_from_Mesh_Rel),
    opp_arg_dat(node_dat1, 3, cell_to_nodes_map, OPP_INC, OPP_Map_from_Mesh_Rel)
);
```

An application developer could write the elemental kernel function `example_kernel` and the `opp_par_loop` declaration as above. Declaring the set and `OPP_ITERATE_ALL` enables the DSL to iterate all elements of that given set.

In this case, the elemental kernel function takes 6 arguments and the loop declaration requires the access method of the data (e.g. `OPP_READ`, `OPP_WRITE`, `OPP_INC`). After the access specifier, `opp_mapping` flag (`OPP_Map_from_Mesh_Rel`) should be provided to all `opp_arg_dats` that need mapping through the particle cell index. In addition, the mapping offset (0,1,2,3) and the `opp_map` mapping should be provided to access the correct node connected to the cell (for the indirectly mapped arguments). Not specifying `opp_mapping` flag and/or a mapping through an `opp_map` indicates that this data should be directly mapped.

API: `opp_par_loop_particle`

```
template <typename... T, typename... OPARG>
void opp_par_loop_particle(void (*kernel)(T *...), char const *name, opp_set set,
                           opp_iterate_type iter_type, OPARG... arguments);
```

Although most of PIC equations can be written as `opp_par_loop` API calls over `particle_set`, `cells_set` or `nodes_set`, particle movement (including handling the change of cell_index of a particle, during inter cell movement) has a different communication pattern. To cater to that requirement, a new API call `opp_par_loop_particle` is introduced to the API.

Similar to `opp_par_loop`, the application developer should implement `opp_par_loop_particle` declarations with similar constructs, however it will only loop over a `particle_set` created using `opp_decl_particle_set`. Nevertheless, the elemental function should always have an `opp_move_var` reference provided as the first argument, which handles the movement routine per particle.

```
struct opp_move_var
{
    bool OPP_iteration_one;
    opp_move_status OPP_move_status;
};
```

`OPP_move_status = OPP_NEED_REMOVE`
: The particle will be removed from the particle set.

`OPP_move_status = OPP_MOVE_DONE`
: The final `cell_index` assigned in the kernel will be set to the particle and the necessary communication of the particle will be handled by the DSL.

`OPP_move_status = OPP_NEED_MOVE`
The same elemental kernel will be called again iteratively with the data corresponding to the new `cell_index` set during the previous elemental function call to the same particle.

The elemental kernel provided to `opp_par_loop_particle` will be iterated until `OPP_move_status` is set to `OPP_NEED_MOVE`. The application developer should compute whether the particle is residing inside the current cell or not, and set the `OPP_move_status` flag accordingly. In addition, the developer could write compute logic specific to iteration one inside a block as stated in the below example.

```
void example2_kernel(opp_move_var& m, ...)
{
    if (m.OPP_iteration_one) {
        // Any computations specific to iteration one
    }

    {
        // Compute logic involving particle and mesh data
    }

    if (is_inside_the_cell) {
        m.OPP_move_status = OPP_MOVE_DONE;
        // Any computate logic, like charge diposition to the final cell or connected nodes
    }
    else if (need_to_remove_from_mesh) {
        m.OPP_move_status = OPP_NEED_REMOVE;
    }
    else { // need_to_search_a_different_cell_in_the_mesh
        m.OPP_move_status = OPP_NEED_MOVE;
        (*cell_index) = calculated_adjoining_cell_index_to_move;
        // Any computate logic, like charge diposition to the passing by cell
    }
}
```

2.2.5 OP-PIC's Other APIs and Utilities

API: `opp_increase_particle_count`

In order to add particles to the simulation, the particle count of the set should be increased, hence the application developer should use the below API call.

```
void opp_increase_particle_count(opp_set particles_set, int num_particles_to_insert);
```

Afterwards, both the `opp_par_loop` and `opp_par_loop_particle` declarations can be used to iterate over the new particles by changing `opp_iterate_type` to `OPP_ITERATE_INJECTED`.

API: `opp_particle_sort`

To gain better particle locality during kernel calls and in applications where double indirection is present (e.g., particle→cell→node), sorting particles according to its residing cell index will be beneficial (after particle injections and particle movements).

```
void opp_particle_sort(opp_set set);
```

However, after calling `opp_particle_sort`, the `OPP_ITERATE_INJECTED` will not iterate any particles at all, since the added particles are no longer considered new to the simulation.

Sorting particles can improve cache usage, but it also introduces overhead. Therefore, it is important to carefully consider when to use it. The user can implement periodic particle sorting using the API provided (we hope to provide the functionality of periodic sorting using a configuration at a later stage). This approach can help achieve the performance benefits of sorting, while minimising the associated overhead costs.

API: `opp_decl_const`

There may be instances where simulation specific constant values are required at elemental kernels (e.g. plasma density). The below API provide flexibility to access these variables at elemental kernels (with the variable name equal to `const_name` in the API), given that they are set after initialising the `OP-PIC` runtime.

```
template <typename T>
void opp_decl_const(int dim, T* data, const char* const_name);
```

API: `opp_reset_dat`

During the simulation there could be occasions where a `opp_dat` needs to reset into a default value (e.g. to zero). The below API call simplifies this by setting the entire `opp_dat` to the `val` provided. For example, an `opp_dat` of dimension 2, will need a `char*` val with data length equal to 2 × size of data type of the `opp_dat` and that will be copied to the entire `opp_dat`.

```
void opp_reset_dat(opp_dat dat, char* val);
```

opp::Params

Configuring an `OP-PIC` simulation is done through a param file, that is essential when running the same simulation with different configuration (example given below). At this stage, only `STRING`, `REAL` (double), `INT` and `BOOL` data types are supported.

```
# Simulation parameters
REAL plasma_den           = 1e16
REAL ion_velocity         = 1e8
REAL electron_temperature = 2e8
REAL wall_potential       = 100000
STRING plasma_species     = Duterium
REAL spwt                 = 2e2
INT max_iter              = 250
REAL dt                   = 1e-12
STRING fesolver_method    = petsc

```

```
# Input files
STRING global_mesh = /ext-home/zl/coarse/mesh.dat
STRING inlet_mesh  = /ext-home/zl/coarse/inlet.dat
STRING wall_mesh   = /ext-home/zl/coarse/wall.dat

# op-pic lib parameters
BOOL opp_auto_sort           = false
INT opp_allocation_multiple = 1
INT opp_threads_per_block   = 64
```

The data can be accessed in the simulation using `params.get<T>(string key)` construct, as given below.

```
    std::string mesh_name = params.get<STRING>("global_mesh");
    double ion_velocity   = params.get<REAL>("ion_velocity");
    bool auto_sort        = params.get<BOOL>("opp_auto_sort");
    int iteration_count   = params.get<INT>("max_iter");
```

3 Porting PIC Applications to OP-PIC DSL

Despite not having a complete unstructured mesh 3D electromagnetic FEM PIC code available to demonstrate the proposed PIC DSL functionality, we have converted three PIC codes to to exhibit the use of API calls and design, with unstructured type indirect data mappings. They are namely,

- SimPIC, an electrostatic 1D FDTD structured mesh PIC code
- CabanaPIC, an electromagnetic 3D FDTD structured mesh PIC code
- Mini-FEM-PIC, an electrostatic 3D FEM unstructured mesh PIC code

The current application implementations for the library can be found at the OP-PIC DSL repository [3]). At this stage, implementations are single node parallelised and do not include MPI parallelisation.

For both SimPIC and CabanaPIC, the structured stencil type computations were converted to unstructured type indirect data mappings (which is loaded from a file prior simulation). The new SimPIC and CabanaPIC codes are serial implementations written in C++ (without MPI) and the calculated particle data and grid point data are verified to be equal to its original implementation.

Mini-FEM-PIC is a sequential electrostatic 3D unstructured mesh FEM PIC example code written in C++ as a part of an online course[4]), that contains an inject particles routine as an addition to the usual PIC algorithm. The DSL converted Mini-FEM-PIC code is based on the optimised Fem-PIC developed as part of this project (see Report 2057699-TN-03-03).

3.1 Porting Mini-FEM-PIC MiniApp to OP-PIC DSL

Mini-FEM-PIC is originally unstructured and the three new implementations (sequential, OpenMP, and CUDA) are written in C++, utilising PETSc [5] (sparse matrix linear solvers) inside the PIC Field Solver, instead of the DSL API calls (the node potential solving calculations need to be broken down to kernels to call the APIs, which will be the focus of future work).

Below, we summarise the steps that should be followed when implementing a PIC application using the `OP-PIC` DSL. In the main C++ function,

(1) Create a `opp::Params` object passing the path to the configuration file.

(2) Load the data from the mesh files (the files can be of user defined type) into C++ arrays.

(3) Initialise the OP-PIC runtime using `opp_init` passing command-line arguments and the param object.

[3]`https://github.com/OP-DSL/OP-PIC`
[4]`https://www.particleincell.com/2015/fem-pic/`
[5]`https://petsc.org/release/overview/`

(4) Define the sets; for Mini-FEM-PIC, cells, nodes, inlet faces and particle sets are required.

(5) Define the mapping between the sets; for Mini-FEM-PIC, `inlet_face_to_cell`, `cell_to_nodes`, and `cell_to_cells` mappings are required.

(6) Define data on sets; create `opp_dat`s for cells, nodes and inlet face sets, passing the data pointers created during the data load-up (OP-PIC will create its own data copy, hence the developer is responsible for cleaning the loaded mesh data). In Mini-FEM-PIC, particles will be injected during the simulations; hence, create `particle_dat`s by passing null pointers.

(7) Make all the constant declarations (using `opp_decl_const`) required at the elemental kernels (e.g. `ion_velocity`).

(8) Start the main loop; start a `std::chrono::system_clock` if runtime of the main loop is needed. Call the API calls for parallel loops and any other computations inside the main loop.

(9) Use `opp_exit` at the end of the simulation. This will ensure all the OP-PIC generated data structures are destroyed and the connections are properly closed.

4 OP-PIC: Mini-FEM-PIC Performance

The test configuration consists of ion flow through a duct. The duct is $2 \times 2 \times 4\ mm^3$ in volume, and is divided up into 7,511 elements (in this case, first-order tetrahedrons). Faces on one end of the duct are designated as inlet faces and the outer wall is fixed at a higher potential (set at $100,000V$) to retain the ions within the duct.

The plasma is fixed at $2 \times 10^8 K$ and the ions are injected with an input velocity of $1 \times 10^8 m/s$. The simulation performs 250 iterations, and particles are weighted back to the grid using a simplistic volume-weighting method. From the start, the ion particle count increases and become steady at around 40 iterations (the ions injected from one end have reached the other end and leave the simulation domain). Five simulation setups are created by changing the ion densities to $1 \times 10^{16}, 1 \times 10^{17}, 2 \times 10^{17}, 4 \times 10^{17}$ and 7×10^{17} ions$/m^3$. This equates to a final ion count of approximately 0.6M, 6M, 12M, 24M and 42M and approximately 15k, 156k, 312k, 624k and 1,092k ions injected per loop, respectively.

The single node CPU runtime is collected from an internal server at the University of Warwick comprising 2 $\times$ Intel Xeon Gold 6252 (Cascade Lake) 2.1GHz 24-core CPUs. The runtime of the $42M$ final particle count setup has additionally been executed on the Sulis system at Warwick, which contains 2 $\times$ AMD EPYC 7742 (Rome) 2.25 GHz 64-core CPUs per node. To test the GPU performance, we use a single NVIDIA P100, NVIDIA V100 or a NVIDIA A100 GPU.

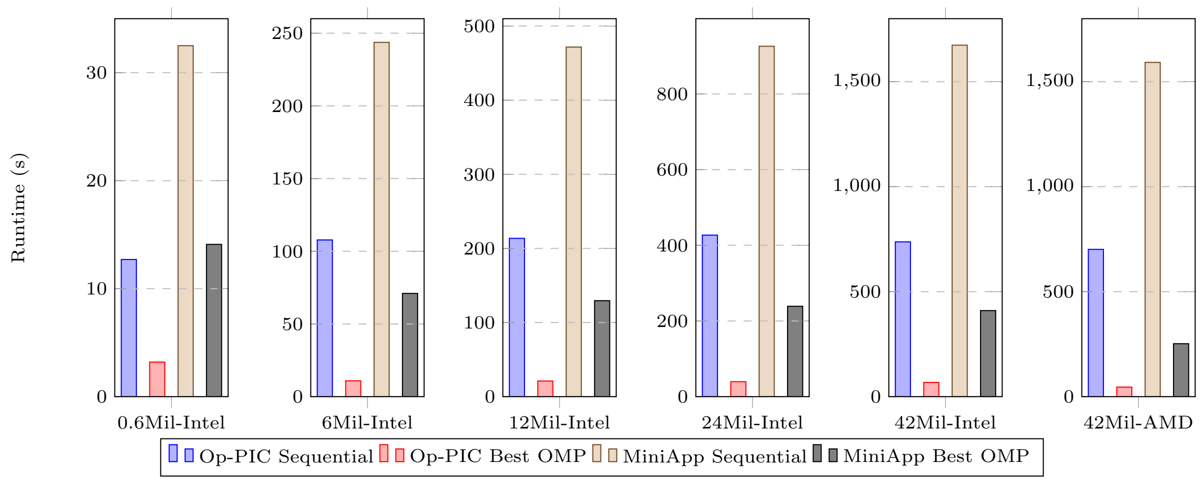

Figure 3: CPU Sequential and Best OpenMP Runtime of OP-PIC DSL and Mini-FEM-PIC MiniApp

According to Figure 3, we can clearly see that the `OP_PIC` best OpenMP is 4× to 15× faster compared with the `OP_PIC` sequential version and in most of the cases, the `OP_PIC` best OpenMP speed-up is around 6× compared with the best OpenMP result of the optimised Mini-FEM-PIC version. In addition, the Mini-FEM-PIC sequential version takes more than twice the runtime of the sequential version of `OP_PIC`, which is unusual (runtimes in seconds can be found in Table 1).

This performance difference may be caused due to more optimised computation pattern in the DSL; these include:

- Data structure layout, we use a Structure-of-Arrays and Mini-FEM-PIC uses an Array-of-Structures data arrangement;
- Data locality and cache usage, we have a different internal logic to handle particle movement/injection (pre-allocating/hole filling/particle sorting);
- Mini-FEM-PIC has OpenMP atomics inside the particle mover, while we use a Kokkos[6] style scatter/-gather method;
- The DSL field solver is running on the PETSc library[7] (this saves a constant time of around 5 seconds for 250 iterations and this is not the dominating factor of the performance difference).

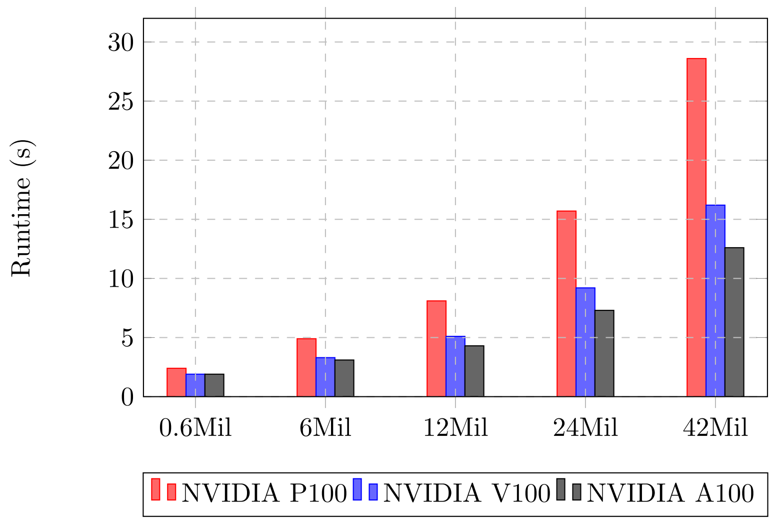

Figure 4: GPU Runtime of OP-PIC DSL

Figure 4 shows the GPU runtime of the OP-PIC DSL converted Mini-FEM-PIC application (there is no GPU support in the Mini-FEM-PIC Mini-App). These results demonstrate that the best GPU results are obtained in an NVIDIA A100 GPU. It is also worth noting that the GPU has great performance improvement compared to CPUs.

[6]https://kokkos.org/about/
[7]https://petsc.org/release/overview/

Final Particle count		**0.6M**	**6M**	**12M**	**24M**	**42M**
Inject per loop		**15k**	**156k**	**312k**	**624k**	**1,092k**
Mini-FEM-PIC Seq	Intel Xeon Gold 6252 CPU	32.5	243.7	471.7	926.150	1,673.8
Mini-FEM-PIC best OMP	Intel Xeon Gold 6252 CPU	14.1	71.0	129.5	238.845	409.8
OP-PIC SEQ	Intel Xeon Gold 6252 CPU	12.7	107.8	213.6	426.811	737.2
OP-PIC best OMP	Intel Xeon Gold 6252 CPU	3.2	10.9	21.1	39.513	67.5
Mini-FEM-PIC SEQ	AMD EPYC 7742 64-Core CPU					1,591.8
Mini-FEM-PIC best OMP	AMD EPYC 7742 64-Core CPU					238.5
OP-PIC SEQ	AMD EPYC 7742 64-Core CPU					701.2
OP-PIC best OMP	AMD EPYC 7742 64-Core CPU					45.4
OP-PIC CUDA	P100 with setup time	2.5	5.7	9.5	18.5	33.5
OP-PIC CUDA	P100 without setup time	2.4	4.9	8.1	15.7	28.6
OP-PIC CUDA	V100 with setup time	2.0	3.9	6.4	11.8	20.8
OP-PIC CUDA	V100 without setup time	1.9	3.3	5.1	9.2	16.2
OP-PIC CUDA	A100 with setup time	1.9	3.8	5.8	10.0	17.3
OP-PIC CUDA	A100 without setup time	1.9	3.1	4.3	7.3	12.6

Table 1: Single node best runtime (in seconds) of OP-PIC DSL and Mini-FEM-PIC MiniApp

References

[1] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation.* Plasma Physics Series. Institute of Physics Publishing, Bristol BS1 6BE, UK, 1991.

[2] John M. Dawson. Particle Simulation of Plasmas. *Reviews of Modern Physics*, 55:403–447, Apr 1983.

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

[4] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1116–1123. IEEE, 2012.

[5] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12, May 2012.

[6] Gihan R Mudalige, Mike B Giles, I Reguly, Carlo Bertolli, and Paul HJ Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12. IEEE, 2012.

[7] István Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran, and Simon McIntosh-Smith. The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations. In *Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 58–67, Washington, DC, USA, 2014. IEEE Computer Society.

[8] B D Dudson, M V Umansky, X Q Xu, P B Snyder, and H R Wilson. BOUT++: a framework for parallel plasma fluid simulations. *arXiv*, physics.plasm-ph:0810.5757, Nov 2008.

[9] Benjamin Daniel Dudson, Peter Alec Hill, David Dickinson, Joseph Parker, Adam Dempsey, Andrew Allen, Arka Bokshi, Brendan Shanahan, Brett Friedman, Chenhao Ma, David Schwörer, Dmitry Meyerson, Eric Grinaker, George Breyiannia, Hasan Muhammed, Haruki Seto, Hong Zhang, Ilon Joseph, Jarrod Leddy, Jed Brown, Jens Madsen, John Omotani, Joshua Sauppe, Kevin Savage, Licheng Wang, Luke Easy, Marta Estarellas, Matt Thomas, Maxim Umansky, Michael Løiten, Minwoo Kim, M Leconte, Nicholas Walkden, Olivier Izacard, Pengwei Xi, Peter Naylor, Fabio Riva, Sanat Tiwari, Sean Farley, Simon Myers, Tianyang Xia, Tongnyeol Rhee, Xiang Liu, Xueqiao Xu, and Zhanhui Wang. BOUT++, 10 2020.

[10] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 676–687. IEEE, 2011.

[11] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, December 2016.

[12] PSyclone Project, 2018. `http://psyclone.readthedocs.io/`.