T/AW087/21
Support and Coordination

Report 2057699-TN-04-01

Proposal for a Particle DSL

Steven Wright and Edward Higgins

University of York

Gihan Mudalige, Zaman Lantra, Ben McMillan, and Tom Goffrey

University of Warwick

February 24, 2023

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 reports we have been focussed on the performance and portability of various programming models and domain specific languages (DSLs) for both fluid and particle methods. In this project we have identified some DSLs that can be used to develop fluid simulations (such as Bout++[1, 2], Nektar++ [3], OPS/OP2 [4, 5, 6], UFL [7]), but have not identified any DSLs focussed on particle methods, where the particles must interact primarily with the mesh, as in PIC.

This report therefore documents our progress towards developing a DSL that can be used for PIC methods, embedded within the OP2 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 [8, 9]. 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.

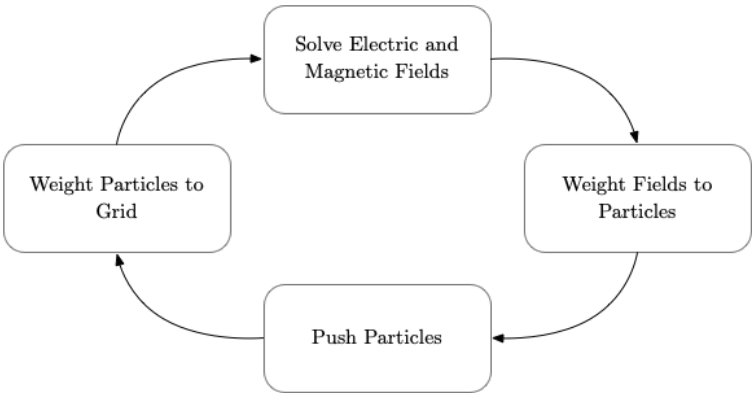

Figure 1: Flow chart summarising the key components of the PIC algorithm

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 (mentioned previously). 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.

We have identified numerous DSLs for developing structured and unstructured mesh computations (e.g. OP2 [5, 10], OPS [4, 6], Bout++ [2, 1], PATUS [11], UFL [7], PSyclone [12], etc.), but none that include support for particle methods. In this report, we detail our progress towards developing an extension to the OP2 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.

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, allowing the mesh to be defined completely and abstractly

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.

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.

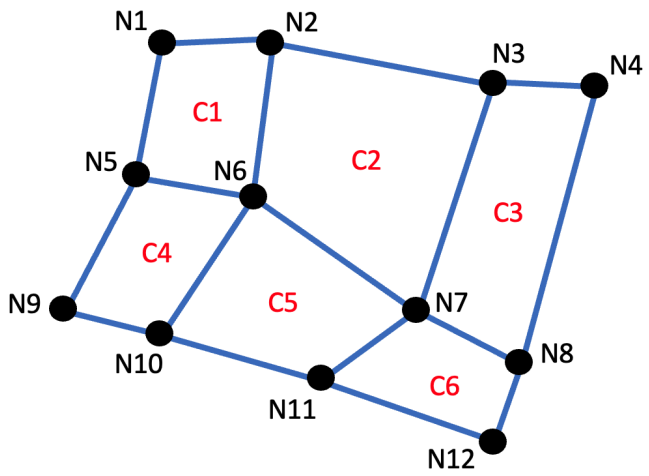

Figure 2: An example unstructured mesh with cells and nodes

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 `op_decl_set`.

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

op_set nodes_set = op_decl_set(n_nodes, "mesh_nodes");
op_set cells_set = op_decl_set(n_cells, "mesh_cells");
```

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

```
op_set particles_set = op_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.

```
op_set op_decl_particle_set(int size, char const *name, op_set cells_set);
```

2.2.2 Defining connectivity (or mapping) between the sets

The connectivity is declared through mappings between sets, using `op_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};

op_map cell_to_nodes_map = op_decl_map(cells_set, nodes_set, NODES_PER_CELL,
                                       cell_to_nodes, "cell_to_nodes_map");

op_map cell_to_cells_map = op_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.

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 `op_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};

op_dat cell_dat1 = op_decl_dat(cells_set, 1, "double", sizeof(double), (char*)d_cell1,
                               "cell field name");

op_dat node_dat1 = op_decl_dat(nodes_set, DIM, "double", sizeof(double), (char*)d_node1,
                               "node field name");
```

The particle dats should be created with `op_decl_particle_dat` and the arguments will be similar to `op_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.

```
op_dat part_dat1 = op_decl_particle_dat(particles_set, 1, "double", sizeof(double),
                                        nullptr, "part field name");

op_dat part_cell_index = op_decl_particle_dat(particles_set, 1, "int", sizeof(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.

```
op_dat op_decl_particle_dat(op_set set, int dim, char const *type, int size, 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: `op_par_loop`

```
template <typename... T, typename... OPARG>
void op_par_loop(void (*kernel)(T *...), char const *name, op_set set,
                 op_iterate_type iter_type, OPARG... arguments);
```

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

```
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 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), since race conditions needs to be considered when executing increments to shared nodes. 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 should provide an optimised code to run on their intended platform.

```
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;
}

op_par_loop(example_kernel, "example_op_par_loop",
    particles_set, OP_ITERATE_ALL,
    op_arg_dat(part_dat1, OP_RW),
    op_arg_dat(cell_dat1, OP_READ, true),
    op_arg_dat(node_dat1, 0, cell_to_nodes_map, OP_INC, true),
    op_arg_dat(node_dat1, 1, cell_to_nodes_map, OP_INC, true),
    op_arg_dat(node_dat1, 2, cell_to_nodes_map, OP_INC, true),
    op_arg_dat(node_dat1, 3, cell_to_nodes_map, OP_INC, true)
);
```

An application developer could write the elemental kernel function `example_kernel` and the `op_par_loop` declaration as above. Declaring the set and `OP_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. `OP_READ`, `OP_WRITE`, `OP_INC`). After the access specifier, a Boolean true value should be provided to all `op_arg_dats` that need mapping through the particle cell index. In addition, the mapping offset (0,1,2,3) and the `op_map` mapping should be provided to access the correct node connected to the cell. Having neither Boolean true and/or a mapping indicates that this data should be directly mapped.

API: `op_par_loop_particle`

```
template <typename... T, typename... OPARG>
void op_par_loop_particle(void (*kernel)(T *...), char const *name, op_set set,
                          op_iterate_type iter_type, OPARG... arguments);
```

Although most of PIC equations can be written as `op_par_loop` API calls over `particle_set`, `cells_set` or `nodes_set`, particle movement (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 `op_par_loop_particle` is introduced to the API.

Similar to op_par_loop, the application developer should implement op_par_loop_particle declarations with similar constructs, however it will only loop over a particle_set created using op_decl_particle_set. Nevertheless, the elemental function should always have an int* move_status given as the first argument, which should be changed to MOVE_DONE, NEED_MOVE, NEED_REMOVE by the application developer within the elemental function.

NEED_REMOVE
: The particle will be removed from the particle set.

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.

NEED_MOVE
: The same elemental kernel will be called again with the data corresponding to the new cell_index set during the previous elemental function call to the same particle.

An example code of an elemental function required for op_par_loop_particle is below.

```
void example2_kernel(int* move_status, int* cell_index, double* ...) {
    {
        // Compute logic involving particle and mesh data
    }

    if (is_inside_the_cell) {
        *move_status = MOVE_DONE;
    } else if (need_to_remove_from_mesh) {
        *move_status = NEED_REMOVE;
    } else { // need_to_search_a_different_cell_in_the_mesh
        *move_status = NEED_MOVE;
        (*cell_index)++; // or compute the most probable cell index to search next
    }
}
```

API: op_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 op_increase_particle_count(op_set particles_set, int num_particles_to_insert);
```

Afterwards, both the op_par_loop and op_par_loop_particle declarations can be used to iterate over the new particles by changing op_iterate_type to OP_ITERATE_INJECTED.

API: op_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 is required (after particle

injections and particle movements).

```
void op_particle_sort(op_set set);
```

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

2.3 OP-PIC: Unstructured Mesh Particle in Cell DSL Implementation

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 & 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
- FemPIC, an electrostatic 3D FEM unstructured mesh PIC code

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 & grid point data are verified to be equal to its original implementation.

FemPIC is a sequential electrostatic 3D unstructured mesh FEM PIC example code written in C++ as a part of the course `https://www.particleincell.com/2015/fem-pic/`, that contains an inject particles routine as an addition to the usual PIC algorithm. FemPIC code is originally unstructured and the new sequential & OpenMP versions are written in C++, utilising PETSc (sparse matrix linear solvers) inside the PIC Field Solver, instead of the DSL API calls (the calculations need to be broken down to kernels to call the APIs, which will be the focus of future work).

The current implementations can be found at `https://github.com/OP-DSL/OP-PIC`. It should be noted that the current implementations include sequential and OpenMP parallelisations, but do not include MPI parallelisation.

References

[1] 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.

[2] 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.

[3] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, and S.J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[4] 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.

[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] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] G.D. Balogh, G.R. Mudalige, I.Z. Reguly, S.F. Antao, and C. Bertolli. Op2-clang: A source-to-source translator using clang/llvm libtooling. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 59–70, 2018.

Corrections/Clarifications

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 [6]) and Firedrake[2] (generating PyOP2 [13]). 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 do not have expertise in these equations, so we will need to work with Physicists/UKAEA scientists]

- *Is an* `op_set` *decomposed over MPI ranks?*

  The `op_set`s exist on each rank and they are just a symbol to denote the collection of dats and maps (particles/nodes/cells etc.) on that rank.

- *"we will keep cell index mapping per particle as data" clarification.*

  In OP2, all the mappings (one `op_set` to another `op_set`) are placed inside `op_map`s, which are static. Even though `cell_index` is a mapping between a `particle_set` to its underlying `cell_set`, the dynamic nature of particles lets us to store the `cell_index` in particles as data, however treat it differently.

- *Cell data - looks like CellDat in NESO-Particles.*

  As of our knowledge, yes. However there could be differences which we are not aware of.

- *Section 2.2.3 particle data - These look like they have a one-to-one mapping with PPMD/NESO-Particles ParticleDat objects*

  There could be differences which we are not aware of. `op_dat`s have the capability of arranging the data in the dat as AOS/SOA depending on the hardware architecture.

- *Is "elemental function" another name for kernel?*

  Yes, we could call it as elemental kernel.

- *op_increase_particle_count: I think there will be a use case for an API where particles can be initialised (e.g. from a particular distribution) then injected*

  As explained in Section 2.2.1, a particle set can be defined by providing the initial particle count (size for the set). This allow the user to initialise particle values for dats, prior to the main loop. As explained in Section 2.2.4, initialisation of injected particles during the main loop can be done using an `op_par_loop` and `op_par_loop_particle` declarations with `iterate_type=OP_ITERATE_INJECTED` (here we could copy data from another dat to particle data if required).

- *op particle sort; If I sort the particles then call a parloop that accesses a grid data set (INC or READ) will the code generation exploit, e.g. combine writes, remove indirections?*

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

Since data on sets are kept per MPI rank, the particle sorting will be done per MPI rank (the particles will belong to the cells of the current MPI rank). The OP-PIC DSL is capable of generating code without data hazards and make halo exchanges when necessary.

- *Please can you give an overview (or point to) how this DSL (embedded in C++?) is construed and code generation occurs (maybe for cases where there is an existing parallel loop implementation)*
  - OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling [14]
  - `https://warwick.ac.uk/fac/sci/dcs/people/gihan_mudalige/talksandpresentations/keynote-europardslaug2022.pdf` (slide 7 & 8)
  - `https://warwick.ac.uk/fac/sci/dcs/people/gihan_mudalige/talksandpresentations/dslworkshoptalk_oct2020.pdf`