UK Atomic Energy Authority

ExCALIBUR

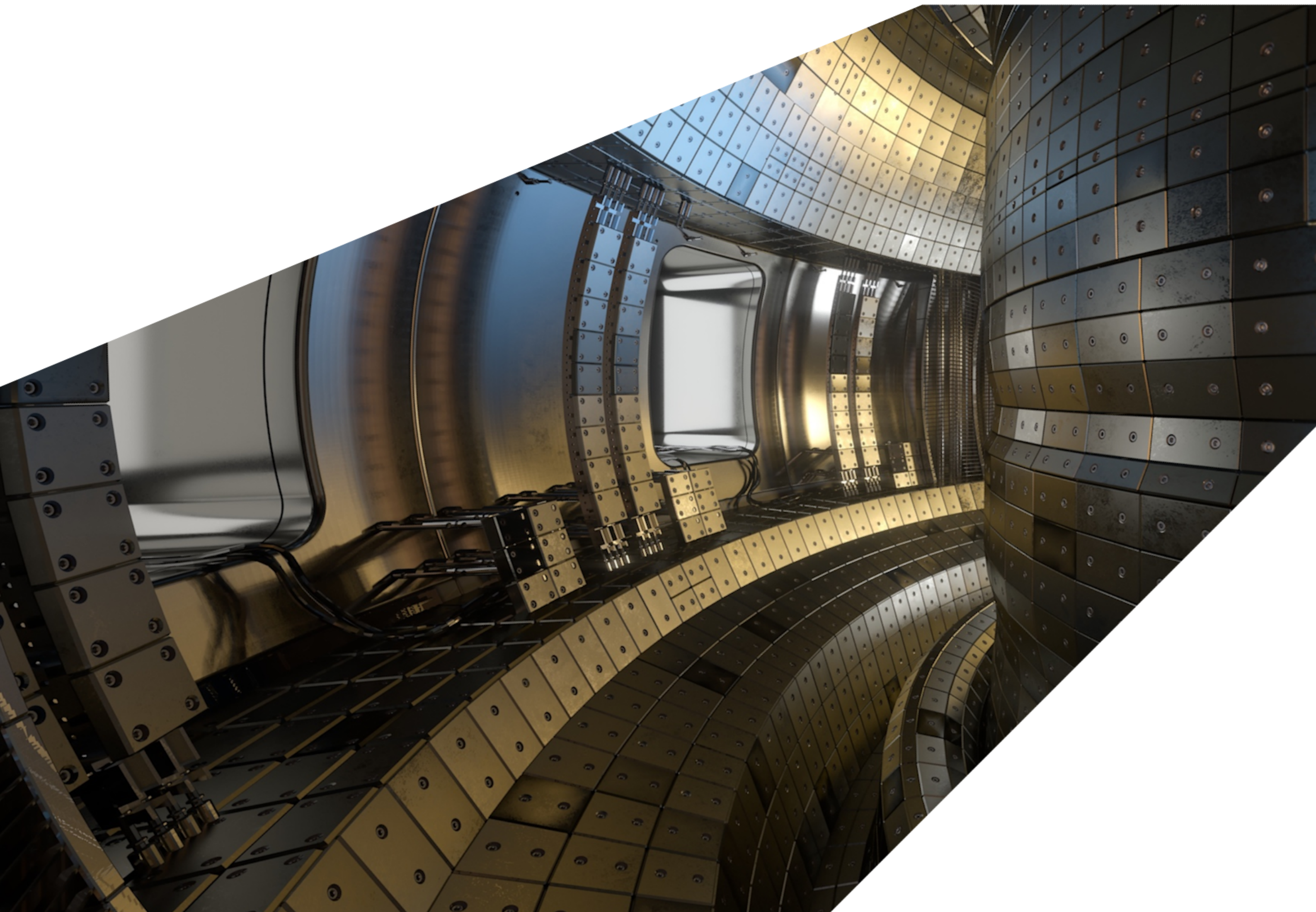
3-D integrated particle and continuum model

M4c.4 Version 1.00

Abstract

This report describes work for ExCALIBUR project NEPTUNE at Milestone M4c.4.

Further developments have been made to the integrated particle and continuum model in “3-D 3-V”, where 3-D 3-V implies that there is variation in three space dimensions (3-D) and three velocity space coordinates (3-V). Nektar++ provides the 3-D finite elements to represent the plasma as a fluid, whereas 3-D 3-V phase space is populated with neutral particles by the SYCL-enabled NESO-Particles library. The calculation is orchestrated by the NESO software, which has been developed under project NEPTUNE. Work reported herein includes: the ParticleLoop implementation in NESO-Particles; the parallel dynamics of the Hasegawa-Wakatani model; and the recombination reaction kernel for neutral particles.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:	
	UKAEA Reference:	CD/EXCALIBUR-FMS/0082
	Issue:	1.00
	Date:	March 15, 2024
Project Name: ExCALIBUR Fusion Modelling System		

	Name and Department	Signature	Date
Prepared By:	Will Saunders James Cook CD	N/A N/A	March 15, 2024 March 15, 2024
Reviewed By:	Ed Threlfall Deputy Principal Investigator		March 15, 2024

1 Introduction

In this report we describe progress made towards the implementation of a 3-D turbulence code tightly coupled to kinetic neutrals. We discuss the augmentation of the implementation of the Hasegawa-Wakatani equations such that the parallel dynamics are accurately incorporated into tightly coupled simulations of turbulence with newly developed neutral physics. Particles interact with the finite-element fluid via an ionisation source term and its reciprocal action, recombination. The particle density is coupled to the plasma density by a time varying source term determined by the spatio-temporal feedback between the neutral particles and the plasma. This report describes the following developments: upgrades to NESO-Particles to use a ParticleLoop, amongst other improvements; a recombination kernel in NESO-Particles.

2 Task Work

2.1 NESO-Particles Additions

2.1.1 ParticleLoop

A “particle loop” is a looping operation where a single “kernel” is applied to all particles in a particular group of particles [1]. Previously we described a high level particle loop interface embedded within the Julia programming language [2]. This Julia implementation closely followed the original PPMD implementation with the computational kernel described by the user as a string literal. This string literal approach is not suitable for the SYCL (C++) NESO-Particles (NP) implementation of the abstraction as it adds additional complexity to consume and manipulate the string to create a kernel that can be passed to the SYCL implementation.

The initial implementation of NP required that users write a particle loop as a SYCL parallel loop. An example of such a loop is presented in Listing 1. This initial implementation has several shortcomings, it requires the user to understand the SYCL programming model and directly interface with particle data via device pointers. It requires a user to write excessive “boilerplate” code to perform relatively simple operations. It is relatively easy to write code that is not correct and is hard to visually inspect for the identification of issues. The code is relatively inflexible, e.g. exactly how the particle loop is executed is difficult to adjust.

Listing 1: Define and launch an old style particle loop to apply a Forward Euler like operation.

```
auto k_P = (*particle_group)[Sym<REAL>("POSITION")]
    ->cell_dat.device_ptr();
auto k_V = (*particle_group)[Sym<REAL>("VELOCITY")]
    ->cell_dat.device_ptr();
auto pl_iter_range = <API to define iteration set>;
auto pl_stride = <API to define iteration set>;
auto pl_npart_cell = <API to define iteration set>;
sycl_target->queue
    .submit([&](sycl::handler &cgh) {
      cgh.parallel_for<>(
          sycl::range<1>(pl_iter_range), [=](sycl::id<1> idx) {
            NESO_PARTICLES_KERNEL_START

            const INT cellx = NESO_PARTICLES_KERNEL_CELL;
            const INT layerx = NESO_PARTICLES_KERNEL_LAYER;

            k_P[cellx][0][layerx] += 0.001 * k_V[cellx][0][layerx];
            k_P[cellx][1][layerx] += 0.001 * k_V[cellx][1][layerx];

            NESO_PARTICLES_KERNEL_END

          });
    })
    .wait_and_throw();
```

There are two main approaches that would extend this particle loop interface with a higher-level user API. The first is a code generation approach where users would describe the particle loop in a higher level language, e.g. Python or Julia, then the lower level code would be generated and compiled. This code generation approach would be motivated by the original PPMD implementation. The second approach, which we applied, is to define a high-level particle loop API directly in C++. By creating a particle loop interface within C++ we eliminate the necessity for users to learn a second programming language. Instead of creating a code generation framework within a high-level language we instead implemented an API within C++ by applying a templating approach.

This template based implementation allows a user to write a particle loop directly within C++. The user does not require detailed knowledge of SYCL to write the loop and neither does one require an intimate knowledge of template metaprogramming. As the API is not dependent on SYCL there is scope to replace SYCL as the computational back-end in future. Furthermore users are abstracted away from the implementation details of the underlying data structures that are accessed by the particle loop. From the point of view of packaging and installing NP, by not using an additional high-level languages users are not burdened with the task of installing an additional programming language on their HPC facility of choice. The trade-off is that the set of available code transformations that can be applied purely within the C++ templating language is more limited than those that can be applied via a code generation framework in Python.

In Listing 2 we describe a particle loop operation that performs an identical operation to the particle loop defined in Listing 1. This particle loop is significantly more approachable than the original implementation. In addition to the improvements for the user facing interface there are additional implementation advantages. The implementation is free to adjust how the particle loop is executed based on the particle distribution presented at runtime. For example in simulations where there is a high degree of anisotropy in the spatial distribution of particle positions the new style particle loops offers a significant speed-up.

Listing 2: Define and launch a new style particle loop to apply a Forward Euler like operation.

```
particle_loop(
    "advection_example",
    particle_group,
    [=](auto P, auto V){
        P.at(0) += 0.001 * V.at(0);
        P.at(1) += 0.001 * V.at(1);
    },
    Access::write(Sym<REAL>("P")),
    Access::read(Sym<REAL>("V"))
  )->execute();
```

Secondly the new interface demands that the user describe exactly how the computational kernel accesses the data structure. For example in Listing 2 the “P” particle data is accessed with a “write” access descriptor whilst the “V” particle data is accessed with a “read” access descriptor. The specification of access descriptors allows the implementation to make optimisations by identifying which data is modified after the execution of a loop. In Section 2.1.2 we describe how these access descriptors modify the behaviour of the loop for non-particle data. Note that this particle loop interface is independent of SYCL and the underlying implementation could be provided by an alternative suitable back-end, e.g. C++ parallel algorithms.

In addition to particle data, which is defined on a per-particle basis, we provide users with additional data structures with which algorithms can be implemented. These data structures are all accessible from the particle loop interface and transparently perform copies between the computational device and the host where required.

The `LocalArray` data structure is an array type that is local to the MPI rank on which it is defined, i.e. no MPI communication occurs. The `LocalArray` type can be considered similar to the `std::vector` type. The `GlobalArray` type is an array type where a copy of the array is stored on each MPI rank and access to the array is collective over the MPI communicator on which the `GlobalArray` is defined. The `CellDatConst` type is a matrix type defined with fixed size (the “Const” part of the name) and data type for each cell in a mesh. A matrix with a fixed number of rows and a variable number of columns per cell can be created per cell with the `CellDat` data structure. Due to the SYCL language restrictions the number of kernel arguments must be fixed at compilation time; the `SymVector` data structure applies indirection to allow a “vector” of particle properties to be passed at runtime. Finally the `ParticleLoopIndex` provides access to indexing information for the particle within the particle loop kernel.

These additional data structures must be passed to a particle loop with an access mode that is commutative. In practice only commutative access modes are defined for these data structures and attempting to pass them to a particle loop with an inappropriate access descriptor will result in a compile time error. We provide an example of the data structure and the intended use in this report, a table of the available access descriptors and the corresponding behaviour is provided in the NP documentation.

2.1.2 LocalArray

The `LocalArray` type is local to each MPI rank. No communication between MPI ranks is performed by the particle loop. The `LocalArray` can be accessed in particle loops with “read” and “add” access modes. In Listing 3 we provide an example use of `LocalArray`.

Listing 3: Example particle loop with `LocalArray` access.

```
// Create a new LocalArray on the same SYCLTarget as the
// particle group.
auto local_array_add = std::make_shared<LocalArray<REAL>>(
  particle_group->sycl_target, // Compute target to use.
  2,                           // Number of elements in the array.
  0                            // Initial value for elements.
);

// Create a local array using initial values from a std::vector.
std::vector<REAL> d0 = {1.0, 2.0, 3.0};
auto local_array_read = std::make_shared<LocalArray<REAL>>(
  particle_group->sycl_target, // Compute device.
  d0                           // Initial values and size definition.
);

auto loop = particle_loop(
  "local_array_example",
  particle_group,
  // LA_ADD is the parameter for the LocalArray with "add" access and
  // LA_READ is the parameter for the LocalArray with read-only access.
  [=](auto ID, auto V, auto LA_ADD, auto LA_READ){

    // Increment the first component of LA_ADD by 1.
    LA_ADD.fetch_add(0, 1);
    // Increment the second component of LA_ADD with the entry in ID[0].
    LA_ADD.fetch_add(1, ID[0]);

    // Read from LA_READ and assign the values to the V particle
    // component.
    V[0] = LA_READ.at(0);
    V[1] = LA_READ.at(1);
    V[2] = LA_READ.at(2);
  },
  // Particle property access descriptors.
  Access::read(Sym<INT>("ID")),
  Access::write(Sym<REAL>("V")),
  // LocalArray access descriptors.
  Access::add(local_array_add),
  Access::read(local_array_read)
);

// Execute the loop.
loop->execute();

// Get the contents of the local array in a std::vector.
auto vec0 = local_array_add->get();
```

2.1.3 GlobalArray

Unlike the `LocalArray`, a `GlobalArray` is intended to have identical values across MPI ranks. The `GlobalArray` can be accessed in “read” and “add” access modes. When accessed with “add” access modes the particle loop must be executed collectively across all MPI ranks. On completion of the loop the local entries of each `GlobalArray` are globally combined (Allreduce). In Listing 4 we provide an example use of `GlobalArray`.

Listing 4: Example particle loop with GlobalArray access.

```
// Create a new GlobalArray on the same SYCLTarget as the particle group.
auto global_array = std::make_shared<GlobalArray<REAL>>(
  particle_group->sycl_target, // Compute target to use.
  1,                           // Number of elements in the array.
  0                            // Initial value for elements.
);

auto loop = particle_loop(
  "global_array_example",
  particle_group,
  [=](auto V, auto GA){
    // Kinetic energy of this particle.
    const REAL kinetic_energy =
      0.5 * (V[0] * V[0] + V[1] * V[1]);
    // Increment the first component of the global array with the
    // contribution from this particle.
    GA.add(0, kinetic_energy);
  },
  // Particle property access descriptor.
  Access::read(Sym<REAL>("V")),
  // GlobalArray access descriptor.
  Access::add(global_array)
);

// Execute the loop. This must be called collectively on the MPI
// communicator of the SYCLTarget as the add operation is collective.
loop->execute();

// Get the contents of the global array in a std::vector. The first
// element would be the kinetic energy of all particles in the
// ParticleGroup.
auto vec0 = global_array->get();
```

2.1.4 CellDat and CellDatConst

The CellDatConst data structure stores a constant sized matrix per mesh cell. When accessed from a particle loop both the “read” and “add” access descriptors expose the matrix that corresponds to the cell in which the particle resides. The CellDat behaves identically to CellDatConst with the extension that the number of rows in the matrix is variable between cells. In Listing 5 we provide an example use of CellDatConst.

Listing 5: Example particle loop with CellDatConst access.

```
// Get the number of cells on this MPI rank.
const int cell_count = particle_group->domain->mesh->get_cell_count();
```

```
// Create a 3x1 matrix for cell_count cells.
auto g1 = std::make_shared<CellDatConst<int>>(
    particle_group->sycl_target, cell_count, 3, 1);

// For each cell get the current matrix (uninitialised as we just
// created the data structure) and zero the values then write back
// to the data structure.
for (int cx = 0; cx < cell_count; cx++) {
  auto cell_data = g1->get_cell(cx);
  cell_data->at(0, 0) = 0.0;
  cell_data->at(1, 0) = 0.0;
  cell_data->at(2, 0) = 0.0;
  g1->set_cell(cx, cell_data);
}

auto loop = particle_loop(
  "cell_dat_const_example",
  particle_group,
  [=](auto V, auto GA){
    // Increment the matrix in each cell with the velocities of
    // particles in that cell.
    GA.fetch_add(0, 0, V[0]);
    GA.fetch_add(1, 0, V[1]);
    GA.fetch_add(2, 0, V[2]);
  },
  // Particle property access descriptor.
  Access::read(Sym<REAL>("V")),
  // CellDatConst access descriptor.
  Access::add(g1)
);

// Execute the loop.
loop->execute();

// After loop execution the 3x1 matrix in each cell will contain
// the sum of the particle velocities in each cell.
for (int cx = 0; cx < cell_count; cx++) {
  auto cell_data = g1->get_cell(cx);
  // use cell data
}
```

2.1.5 Vectors of Particle Properties

Particle properties are stored within the `ParticleGroup` in `ParticleDat` instances. `SymVector` enables passing a set of `ParticleDat` instances to the `ParticleGroup` where the number of `ParticleDat` instances is only known at runtime. The indexing of the data structure in the kernel is via the positional index of the `ParticleDat` relative to the construction of the `SymVector`. In Listing 6 we provide an example use of `SymVector`.

Listing 6: Example particle loop with `SymVector` access.

```
// These constants are captured by value into the kernel lambda.
const int ndim = 2;
const REAL dt = 0.001;

auto loop = particle_loop(
  "sym_vector_example",
  particle_group,
  [=](auto dats_vector){
    for(int dimx=0 ; dimx<ndim ; dimx++){
      // P has index 0 in dats_vector as it is first in the sym_vector.
      // V has index 1 in dats_vector as it is second.
      dats_vector.at(0, dimx) += dt * dats_vector.at(1, dimx);
    }
  },
  // We state that all ParticleDats in the SymVector are write access.
  Access::write(
    // Helper function to create a SymVector.
    sym_vector<REAL>(
      particle_group,
      // This argument may also be a std::vector of Syms.
      {Sym<REAL>("P"), Sym<REAL>("V")}
    )
  )
);

// Execute the loop.
loop->execute();
```

2.1.6 Particle Indexing Helper Functions

Some data structures are indexed by the cell and layer of the particle. The `ParticleLoopIndex` is a data structure that can be read in a `ParticleLoop` to provide this information. This data structure is read-only. In Listing 7 we provide an example use of `ParticleLoopIndex`.

Listing 7: Example particle loop with `ParticleLoopIndex` access.

```
auto loop = particle_loop(
  "loop_index_example",
  particle_group,
  [=](auto index){
    // Dummy output variable we store the indices in for this example.
    [[maybe_unused]] INT tmp;
    // The cell containing the particle.
    tmp = index.cell;
    // The row (layer) containing the particle.
    tmp = index.layer;
    // The linear index of the particle on the calling MPI rank.
    // This index is in [0, A->get_npart_local()).
    tmp = index.get_local_linear_index();
    // The linear index of the particle in the current ParticleLoop.
    // This index is in [0, <size of ParticleLoop iteration set>).
    tmp = index.get_loop_linear_index();
  },
  // Note the extra {} that creates an instance of the type.
  Access::read(ParticleLoopIndex{})
);

loop->execute();
```

2.1.7 Particle Products

When modelling atomistic processes it is necessary to be able to create and destroy particles. In particular a process may create new child particles from an existing parent particle. These child particles may inherit a subset of the parent particle properties whilst generating new values for the remaining properties. The `ProductMatrix` and `DescendantProducts` types are data structures designed as an intermediate staging area for particle properties from which new particles should be created.

We consider a scenario where N existing particles are to each create M new particles. These new particles will ultimately be added to the `ParticleGroup` that contains the original N particles and hence each of the NM new particles require all of the properties to be set to sensible values. The particle properties can always be modified after the new particles are added to the `ParticleGroup` by using a `ParticleLoop` therefore here we discuss methods to define particle properties before the particles are added.

Particles are created via the `DescendantProducts` data structure that provides space for each of the new particle properties. These particle properties are accessed from each of the N parent particles from a `ParticleLoop` kernel. Once a `DescendantProducts` instance is populated with values those values can be used to create new particles in the parent `ParticleGroup` by calling “add_particles_local” with the new properties.

When a new `DescendantProducts` instance is created the number of output products M per existing particle is specified along with the REAL and INT particle properties that will be explicitly set in the `ParticleLoop` kernel. This kernel must also set the parent particle of the new product particle by calling “set_parent” for the product particle to be included when `add_particles_local` is called.

The `set_parent` mechanism provides two functions, firstly it defines the product as a product that should be included when `add_particles_local` is called. This `set_parent` call can be intentionally omitted to mask off any number of the M products that ultimately were not required. Secondly for particle properties that are not defined in the `DescendantProducts` constructor the `set_parent` call defines the parent particle from which these properties should be copied.

The destination `ParticleGroup` for the new particle products has a set of particle properties for each of the particles in the `ParticleGroup`. When the `DescendantProducts` are are added via `add_particles_local` there are two options. If the property is explicitly defined in the `DescendantProducts` instance then the component values, for all particles and all components of that property, are copied from the `DescendantProducts` into the corresponding `ParticleDat` in the `ParticleGroup`. If a property is defined in the `ParticleGroup` and not in the `DescendantProducts` then for all new particle entries in the `DescendantProducts` the component values for that property are copied from the parents specified in the `DescendantProducts`.

Note that the decision to copy property values from a parent particle or from a `DescendantProducts` entry is taken property wise for all particles for all components of the property. If finer control is required, e.g. to inherit particle properties on a per particle or per component level, then the user should specify the property in the `DescendantProducts` instance and populate the new properties for all particles and components in the `ParticleLoop` kernel that is responsible for populating the entries in the `DescendantProducts` instance.

In the `ParticleLoop` kernel that sets the properties of the new products the property, and component, are specified by an integer index instead of the `Sym` objects used in host code. The ordering of the properties is defined as the order in which particle properties are specified for the `DescendantProducts` specification. This ordering is contiguous within all properties of the same data type, e.g. if two INT properties are specified then the integer properties are indexed with 0 and 1 and if three REAL properties are specified then the real valued properties are indexed 0,1 and 2. Interlacing of INT and REAL properties in the specification is ignored, the properties are indexed within the set of properties that have the same type in the order they are specified. In Listing 8 we provide and example of the API for creating new child particles.

Listing 8: Creating new child particles from parent particles.

```
/* For this example we assume that the particle group has the
   following REAL and INT properties with the specified number
   of components:

   REAL
   ----
    P 2 (positions)
    V 3
    Q 1

   INT
   ---
```

```
    CELL_ID 1 (cell id)
    ID 1
  */

// We create a DescendantProducts with the following specification.
// The number of components for a property must match the number of
// components in the ParticleGroup the products are added to.
auto product_spec = product_matrix_spec(
  ParticleSpec(
    ParticleProp(Sym<REAL>("V"), 3),
    ParticleProp(Sym<REAL>("Q"), 1),
    ParticleProp(Sym<INT>("ID"), 1)
  )
);

/* Re-visiting the properties in the particle group and the product
   specification:

  REAL
  ----
  P: Is a ParticleGroup property not in the product spec
     therefore the values will be copied from the parent particles.
  V: Is defined in the product spec and the values set in the
     DescendantProducts will be used for the new particles.
  Q: Is defined in the product spec and the values set in the
     DescendantProducts will be used for the new particles.

  INT
  ---
  CELL_ID: Is a ParticleGroup property not in the product spec
           therefore the values will be copied from the parent particles.
  ID: Is defined in the product spec and the values set in the
  DescendantProducts will be used for the new particles.
*/

// Create a DescendantProducts with the above product spec for at most 2
// products per parent particle.
const int num_products_per_particle = 2;
auto dp = std::make_shared<DescendantProducts>(
  particle_group->sycl_target,
  product_spec,
  num_products_per_particle
);

// Define a ParticleLoop which creates the products from the parent
// particles.
```

```
auto loop = particle_loop(
  particle_group,
  [=](
    auto DP, auto parent_index, auto V, auto Q, auto ID
  ){
    for(int childx=0 ; childx<num_products_per_particle ; childx++){
      // Enable this product by calling set_parent
      DP.set_parent(parent_index, childx);

      // The V property was the first REAL product we specified
      // and therefore has property index 0 for at_real.
      const int V_index = 0;
      for(int dimx=0 ; dimx<3 ; dimx++){
        // Copy V from parent but negate the sign.
        DP.at_real(parent_index, childx, V_index, dimx) =
            -1.0 * V.at(dimx);
      }

      // The Q property was the second REAL product specified
      // and hence has property index 1 for set_real.
      const int Q_index = 1;
      // Simply copy the parent Q value in this kernel.
      DP.at_real(parent_index, childx, Q_index, 0) = Q.at(0);

      // The ID property is the first INT property we specified
      // and hence has index 0 for at_int.
      const int ID_index = 0;
      // Copy parent ID but modify it.
      DP.at_int(parent_index, childx, ID_index, 0) = -1 * ID.at(0)
        - 100 * childx;
    }
  },
  Access::write(dp),
  Access::read(ParticleLoopIndex{}),
  Access::read(Sym<REAL>("V")),
  Access::read(Sym<REAL>("Q")),
  Access::read(Sym<INT>("ID"))
);

// Before a loop is executed that accesses a DescendantProducts
// data structure the reset method must be called with the
// number of particles in the iteration set of the loop.
dp->reset(particle_group->get_npart_local());

// Execute the loop to create the products.
loop->execute();
```

```
// Finally add the new products to the ParticleGroup.
particle_group->add_particles_local(dp);
```

2.1.8 ParticleSubGroup

The NumPy project is the de-facto library for array operations in the Python ecosystem. A useful feature NumPy offers is the ability to create “views” into existing arrays. A view into an array presents a subset of the original array as a new array instance without copying elements. Selection of this subset of elements may be non-trivial and cumbersome to use. By creating a view the process of applying an operation to a subset of the elements is simplified for the user without creating additional copies of the data.

In NP we offer the `ParticleSubGroup` class that selects a subset of the particles held in a parent `ParticleGroup`. A user creates a sub group by specifying a conditional that evaluates to true for all particles in the sub group and false for those that are not. This conditional is a read-only function of particle data. On creation of a sub group the implementation records the indices of particles for which the conditional holds, no particle data is copied to create the sub group. The implementation uses the access descriptors we described in Section 2.1.1 to determine when the stored representation of the sub group has been invalidated by the modification of particle data or other events, such as moving particles between cells or MPI ranks.

Listing 9: Create a new `ParticleSubGroup` containing particles with even particle ids then apply a Forward Euler like operation to only those particles.

```
// Create a ParticleSubGroup from even values of ID.
auto sub_group = particle_sub_group(
  particle_group,
  [=](auto ID) {
    return (ID[0] % 2 == 0);
  },
  Access::read(Sym<INT>("ID"))
);

// Perform a position update style kernel on particles
// with even values of ID[0].
auto loop = particle_loop(
  sub_group,
  [=](auto V, auto P){
    P[0] += 0.001 * V[0];
    P[1] += 0.001 * V[1];
  },
  Access::read(Sym<REAL>("V")),
  Access::write(Sym<REAL>("P"))
);
```

```
loop->execute();
```

A sub group may be used in place of a particle group in numerous ways. Most interestingly a sub group is a valid iteration set for a particle loop operation. In Listing 9 we demonstrate the definition of a particle sub group that contains particles with an even value in a hypothetical “ID” particle integer property. We also demonstrate the application of a particle loop that executes over the particles in this sub group.

A user could implement the example in Listing 9 with a single particle loop that combines the conditional that creates the sub group and the Forward Euler kernel. The sub group is computationally advantageous when a subset of a particle group is selected, in a particle sub group, then this sub group is reused for multiple subsequent operations. An efficient interface exists to create a sub group that contains every particle in the parent particle group. This functionality allows users to create generic interfaces that expect to be passed sub groups instead of particle groups.

Sub groups may be created from sub groups allowing a hierarchy of views into a single parent particle group to be created. Furthermore a user may specify which particles are to be removed from a particle group or alternatively added to a different particle group by passing a sub group.

An example use case would be to first identify particles in the simulation that have very small computational weight then combine these particles into a single particle per cell. This amalgamation process can be achieved using the additional data structures we described in Sections 2.1.2-2.1.6 where the exact process will depend on how the user designs the process to occur.

2.2 NESO Composite-Particle Interaction Interface

Our existing numerical experiments, that contain particles, have applied simplistic periodic boundary conditions in computational domains where identifying the particles that have left the domain is computationally simple (i.e. working with cuboids is easy). The boundary of a cuboid domain is exactly representable by a linear 3-D mesh, with this property we can simply compare particle positions with the cuboid origin and extents to determine if a particle has crossed a boundary.

If we consider a more complex target geometry, for example a torus, there is a discrepancy between the desired computational domain and the discrete approximation achieved by the mesh. In Figure 1 we present a coarse approximation of a torus constructed from a collection of tetrahedra. In this torus geometry it is insufficient to determine if a particle has crossed the domain boundary by comparing the particle position with the original, i.e. pre-meshing, definition of the torus. With a more complex domain it is unrealistic to compare particle positions with the original CAD description.

In practice it is highly desirable for a user to specify different boundary conditions for each surface in the computational domain. For example consider a cylinder domain with particle flow along the axis of the cylinder. To model the relevant physical system the user may specify that the particles that cross the ends of the domain are deleted and that particles that cross the sides of the domain are reflected back into the domain.

The extensive set of possible boundary conditions a user may apply motivates a library design where the list of particles that crossed each boundary in a move operation are provided to the

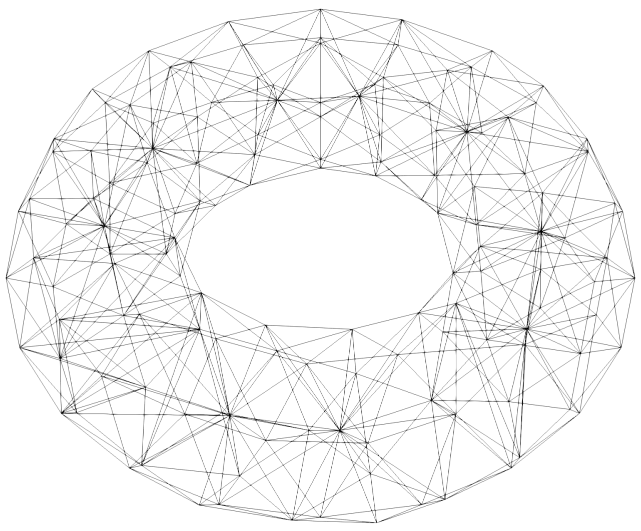


Figure 1: Wireframe of torus mesh composed of tetrahedra.

user and it is the responsibility of the user to describe exactly how these particles are treated. In *Nektar++* boundary conditions, for the finite element method, are specified on a per "composite" basis where a composite is a collection of elements of the same type. For example in the torus presented in Figure 1 the composite with id "100" is the set of triangles that form the surface.

In NP we provide the particle sub group class as a lightweight view into a particle group formed from all particles that satisfy a condition. The composite interaction capability in NESO, which we will now describe, creates a map from composite ids, which were specified by the user, to the sub group of particles that passed through the composite in the last time step. As particle sub groups are permissible iteration sets for particle loops, a user describes the physical process to apply on each boundary in terms of particle loops. In Figure 2 we present a snapshot of a system where particles are transported inside a torus and are reflected upon contact with the torus wall.

The NESO implementation provides the set of particles that intersect each composite as a particle sub group, using this functionality we implement an algorithm to apply a reflection process to each particle. This reflection process, and corresponding advection operator, is an example of a process where a set of operations is iteratively repeated at each time step until all particles in the system have completed a time step. In Algorithm 1 we present the high-level overview of a time step with reflective boundary conditions. Algorithm 1 assumes that there is an implementation to detect and compute the intersection of particle trajectories and mesh composites.

For the implementation we describe in this report, we assume that the mesh is linear and hence all the 2-D mesh elements that form the surface of the 3-D mesh each exist in a plane. To determine if a line segment intersects a given triangle or quadrilateral we first determine the intersection of the line segment and the plane that contains the element. Once that intersection point is computed we determine if the point lies within the geometry object itself by repurposing the Newton iteration implementation that exists to bin particles into mesh elements.

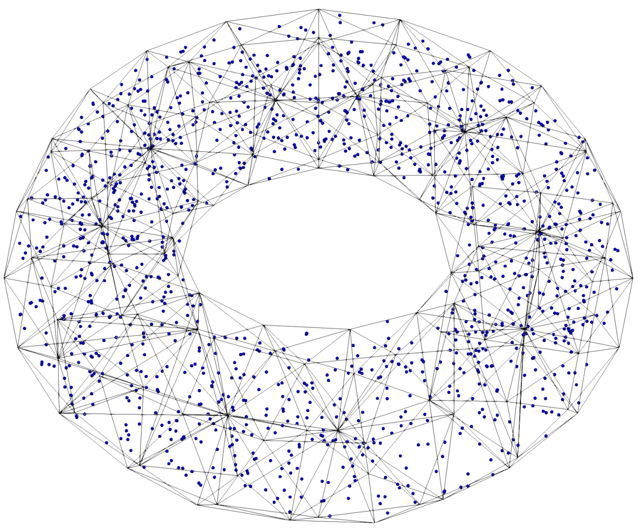


Figure 2: Wireframe of torus mesh composed of tetrahedra containing particles.

To evaluate the Newton iterations required to determine if a point lies within a surface element we require the description of the surface element itself. As is typical of MPI parallelised FEM simulation, the mesh itself has undergone a domain decomposition process and is distributed across the set of MPI ranks. We assume that for a scientifically interesting simulation it is impracticable, due to memory constraints, to store all surface elements on all MPI ranks. Secondly we require an implementation that correctly accounts for the scenario where a particle travels over an MPI decomposition boundary, i.e. $\vec{r}_i$ and $\vec{r}'_i$ are points on different MPI ranks, and in this one step the trajectory intersects a surface element owned by a remote MPI rank.

To collect the elements with which a particle trajectory potentially intersects, we use the coarse Cartesian mesh described in [3]. This coarse mesh has the property that each cell is uniquely owned by an MPI rank and this owning rank is known globally. We build a map from each Cartesian cell to the 2-D surface elements that intersect the Cartesian cell. By determining the Cartesian cells intersected by the line segment we determine the candidate 2-D surface elements to inspect for intersections with the particle trajectory. In Algorithm 2 we provide a high-level description of the setup process required to map Cartesian cells to 2-D surface elements. Secondly, in Algorithm 3 we provide an overview of the process to determine if a particle trajectory intersects a composite.

```
Set $t_i \leftarrow 0$
Set $t'_i \leftarrow 0$
Set $a \leftarrow$ ParticleSubGroup($i \in A$)
while $|a| > 0$ do
    for particle $i \in a$ do
        Set $\delta t_i = \delta t - t_i$
        Set $\vec{r}'_i \leftarrow \vec{r}_i$
        Set $\vec{r}_i \leftarrow \vec{r}_i + \delta t_i \vec{v}_i$
        Set $t_i \leftarrow t_i + \delta t_i$
        Set $t'_i \leftarrow \delta t_i$
    end
    $b \leftarrow$ ParticleSubGroup($i \in a$ where the line segment between $\vec{r}'_i$ and $\vec{r}_i$ intersects a
      surface composite)
    for particle $i \in b$ do
        Read $\vec{c}_i$, the intersection of the segment between $\vec{r}'_i$ and $\vec{r}_i$ and the surface
          composite
        Read $\vec{n}_i$, the normal to the surface at $\vec{c}_i$
        Set $\vec{v}_i$ from reflection computed with surface normal $\vec{n}_i$
        Compute proportion of time step completed before intersection with surface
          $p = \frac{|\vec{c}_i - \vec{r}'_i|}{|\vec{r}_i - \vec{r}'_i|}$
        Set $\vec{r}_i \leftarrow \vec{c}_i$
        Set $t_i \leftarrow t_i + (p - 1)t'_i$
        Set $t'_i \leftarrow pt'_i$
    end
    Set $a \leftarrow$ ParticleSubGroup($i \in b$ where $t_i < \delta t$)
end
```

Algorithm 1: Overview of algorithm to apply reflective boundary conditions to a system of particles stored in a particle group A. Where for particle i, $\vec{r}_i$ is the position, $\vec{r}'_i$ is the previous position, $\vec{v}_i$ is the velocity, $t_i \in [0, \delta t]$ records the progression through each time step of size δt, t'_i records the time increment of the last advection operation.

```
for Element $e \in H$ do
    Compute bounding box $b$ of $e$
    for Cartesian cells $c$ which intersect $b$ do
        Determine owning rank $r_c$ of Cartesian cell $c$
        Push $e$ onto list of elements to send to $r_c$
    end
end
Exchange 2-D element description lists. Such that the MPI that owns cell $c$ holds all the
  2-D elements that intersect $c$
```

Algorithm 2: Overview of algorithm to build map from Cartesian cells to 2-D elements that form composites of interest. Where H is a set of composite indices of interest. This algorithm is performed on each rank r and is collective across all MPI ranks in the communicator.

```
Reset list of Cartesian cells $C$ to collect elements for.
for particle $i$ do
    Determine Cartesian cell $c$ containing $\vec{r}_i$
    Determine Cartesian cell $c'$ containing $\vec{r}'_i$
    for all cells $\hat{c}$ between $c'$ and $c$ do
        Append $\hat{c}$ to $C$
    end
end
for cell $c$ in $C$ do
    if $c \notin D$ then
        Retrieve all 2-D elements from the MPI rank that owns $c$
        Append $c$ to $D$
    end
end
for particle $i$ do
    Determine Cartesian cell $c$ containing $\vec{r}_i$
    Determine Cartesian cell $c'$ containing $\vec{r}'_i$
    Set $d \leftarrow \inf$ (maximum floating point value in practice)
    Initialise intersection point $\vec{c}_i$
    for all cells $\hat{c}$ between $c'$ and $c$ do
        for elements $e \in \hat{c}$ do
            Compute intersection point $\vec{c}_e$ of the line segment between $\vec{r}'_i$ and $\vec{r}_i$ and the
              plane containing $e$.
            if $\vec{c}_e$ exists then
                if $\vec{c}_e \in e$ then
                    Set $d_e \leftarrow |\vec{c}_e - \vec{r}'_i|$
                    if $d_e < d$ then
                        Set $d \leftarrow d_e$
                        Set $\vec{c}_i \leftarrow \vec{c}_e$
                    end
                end
            end
        end
    end
end
```

Algorithm 3: Overview of algorithm to detect intersection between particle trajectories and mesh composites. Where for each particle i: $\vec{r}_i$ is the current particle position, $\vec{r}'_i$ is the previous particle position. D is a list of Cartesian cells that have been previously inspected and for which this rank holds copies of all the element information.

The intersection detection algorithm described in Algorithm 3 is designed to be robust for the case where a particle travels through multiple surface elements in a single step, e.g. by passing through a corner. The process is collective on the MPI communicator as at an intermediate stage the descriptions of remotely owned elements are collected. These elements are cached for later use to avoid repeated communication of element information. Furthermore the communication procedure begins with a global vote to determine if any further communication must occur. It is expected that in a relatively small number of time steps each rank will contain all required 2-D elements in the cache and little further communication will be required.

2.3 Grantee Work

Part of the collaboration with the University of Warwick and University of York investigates the development of a Domain Specific Language (DSL) for particle based operations. The reports [4, 5] propose a DSL for describing particle operations and begin to investigate the performance of the implementation in select proxy applications.

The proposed DSL formulates the particle loop abstraction within the existing OP2 [6] implementation. The extension which adds the particle loop abstraction to the OP-DSL ecosystem is developed under the name “OP-PIC”. Within the OP-DSL ecosystem, parallel loops are executed over iteration sets described by sets of data. Furthermore, maps may be described between different sets. For example there may exist a set which indexes mesh elements and a set which indexes the mesh vertices. A map can then be constructed from mesh element to vertices which touch the mesh cell. Data can be assigned to each element of a set in a so called “dat”.

In the OP-DSL ecosystem a parallel loop is executed over a set, for example the set of mesh elements. For each element in the iteration set a computational kernel is executed which may access the associated data and maps for the element in the set. These kernels must be suitable for parallel execution. In the OP-PIC extension a set of particles may be defined and data is assigned to each particle. These particle sets are valid iteration sets for an OP-PIC parallel loop. With these primitive types and loops the particle loop is now implemented within the OP-DSL ecosystem.

By defining maps from sets of particles to mesh elements the OP-PIC framework allows users to describe how the data movement must occur to describe the conversion from particle based representations to mesh based representations. For example a charge deposition process onto a DG0 function space, where each cell contains a constant valued function, can be performed by creating the map mesh cell to particles contained within the cell. Each cell is then assigned a “dat” which holds the single DOF per cell. The charge deposition loop then loops over each cell and traverses the map from cells to particles where then the DOF for the cell is incremented with the contribution for each particle within the cell.

For each described particle loop the proposed implementation would use the OP-PIC framework to generate the source code which performs the looping operation requested by the user. OP-PIC may then generate different implementations for different architectures as required. The OP-PIC framework must provide particle movement implementation to move particles over the decomposed mesh.

The particle movement operation is a non-trivial process to efficiently and robustly implement. To move particles between nearby MPI ranks the OP-PIC framework implements a “multi-hop” process to identify the new MPI rank which owns a particle when it leaves the portion of mesh owned by the previous MPI rank. This multi-hop process is extended with a “direct-hop” process which, like NESO-Particles, adds an additional Cartesian mesh over the unstructured mesh to facilitate the movement of particles which take large steps in space per time step.

The reports present timing results from the OP-PIC implementation where the authors compare a charge deposition process on multiple CPU and GPU architectures. This charge deposition process is an interesting test case as an efficient implementation must overcome a parallel write contention when many particles contribute to a single mesh DOF. The optimal way to perform this reduction is architecture dependent. Furthermore the report presents weak scaling results from CPU and GPU platforms to demonstrate the weak scaling performance of the particle movement process implemented in OP-PIC.

2.4 Strong Scaling Experiments

As part of the software development process for HPC it is important to continuously evaluate and analyse the performance of the implementation. In previous strong scaling studies [7] we established that the strong scaling limit, i.e. the smallest problem size for which good parallel efficiency is maintained, was approximately 10^5 particles per CPU core for a two-stream instability example.

To ensure we maintain at least this level of performance we constructed a micro-benchmark which transports a large number of particles over a square domain constructed as a Cartesian mesh. This Cartesian mesh has the feature that mapping particle positions is relatively cheap and hence most of the runtime of the benchmark is in the particle bookkeeping routines. The bookkeeping routines are common to all simulations which use NP and hence the performance of these implementations is important.

In Figure 3 we present the results of a strong scaling experiment on the ARCHER2 HPC facility. These results use AdaptiveCPP (OpenMP library back-end, GCC 11.2.0, cray-mpich 8.1.23) as the SYCL implementation. Although this SYCL implementation is OpenMP based, we use a single OpenMP thread per CPU core. With this configuration we launch 4096 MPI ranks when running the experiments on 32 nodes. The size of the computational problem was chosen to reach approximately 10^5 particles per core when 16 nodes are used. ARCHER2 is a HPE Cray EX system where computational nodes are comprised of two EPYC 7742 64-core 2.25GHz processors. These results demonstrate good parallel efficiency (90%+) with 10^5 particles per core.

2.5 Recombination

A natural next addition to the capabilities of the NESO software is to add the leading opposing process to electron-impact ionisation, namely radiative recombination[8]:

$$e + H^+ \rightarrow H + h\nu, \tag{1}$$

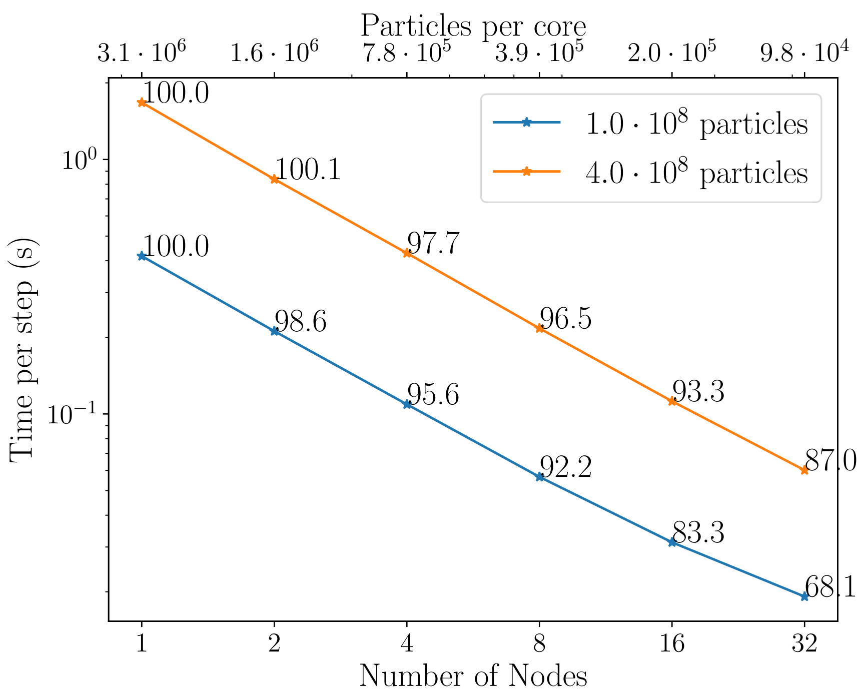


Figure 3: Strong scaling advection experiment on ARCHER2. In the top, orange, line 4×10^8 particles are transported over a Cartesian mesh with 512 cells in each dimension. In the bottom, blue, line 1×10^8 particles are transported over a mesh with 256 cells in each dimension. Note that both experiments feature the same number of particles per mesh cell. In-figure numbers are the parallel efficiency relative to a single node. Top figure axis lists the particles per core for the experiment with 4×10^8 particles.

where strictly speaking the resulting Hydrogen atom is a function of the primary quantum number n. However, in this proof-of-principle work we take the assumption that $n = 1$ and that it is safe to ignore $h\nu$.

The derivation of the rate coefficients for Maxwellian electron and proton distributions follows naturally by applying the detailed balance principle [8]: i.e. the rate coefficient is such that local thermal equilibrium is maintained in the presence of ionisation. The expression for the recombination rate, α_{rec}, is

$$\alpha_{rec}(nl) = \frac{A_{nl}}{n} \frac{\beta_b^{3/2}}{\beta_n + \xi_{nl}}(\beta_n) \tag{2}$$

where $\beta_n = I_n/T$, $I_n = R_y/n^2$, $\xi_{(n=0,l=s)} = 0.25$, is the Bohr radius, $R_y = 13.6$ eV is the Rydberg constant.

In this report, we continue to use the implementation of the electron-impact ionisation process as detailed in the last milestone M4c.3. It was the first reaction process between the fluid-plasma and the particle-neutrals and we refer the reader back to the previous report [9] for further information. A discussion of the implementation details behind the newly developed the interaction capability follows in the next section.

2.5.1 The recombination ParticleLoop

Developments made to the NESO-Particles performance portable particle library, as described above, have brought easier and more robust methods to develop particle algorithms on accelerator style devices like GPUs. In this subsection we show one code snippet, used in NESO, to elucidate both the algorithm behind recombination and its implementation.

Referring to Listing 10, lines 2-7 are dedicated to calculating quantities needed in the loop or deliberately making copy assignments of other variables. These are all constant values, which may then be used inside the lambda function. Line 11 is where the `ParticleLoop` begins and its first three arguments are: a string to name the loop for profiling purposes on line 12; the `ParticleGroup` object that the Loop will traverse and operate upon on line 13; and the lambda function itself beginning on line 14, which encapsulates the logic to applied to the Particles in the `ParticleGroup`. The final arguments on lines 24-27, inclusive, are the access descriptors to the `ParticleDat` instances that correspond to the selfsame ordered arguments to the lambda function on line 14. The access descriptors define the type of the argument and whether or not it will be mutated. The logic of the `ParticleLoop` in lines 15-21 simply calculates the reaction rates and the weight of the newly created recombined neutral particle and the amount of plasma density that is duly lost in the process. Finally on line 30, the `ParticleLoop` is executed and the `ParticleGroup` is operated upon according the logic defined in the lambda function. After execution, the `ParticleDat` instances will be filled with the appropriately updated values required for logic elsewhere in the program.

2.5.2 Turbulence simulations with ionisation and recombination physics

In this section we present simulations of 3-D fluid turbulence using higher order finite elements combined with GPU capable neutral particle physics. The plasma turbulence model that we use is the Hasegawa-Wakatani model with an additional source term for plasma density, S_n, arising from the evolution of the neutral particles. These equations for the plasma density n and potential ϕ are

$$\frac{\partial n}{\partial t} = \{n, \phi\} - \kappa \frac{\partial \phi}{\partial y} + \alpha \nabla_{\parallel}^2 (n - \phi) + S_n, \tag{3}$$

$$\frac{\partial \zeta}{\partial t} = \{\zeta, \phi\} + \alpha \nabla_{\parallel}^2 (n - \phi), \tag{4}$$

where $\zeta = \nabla_{\perp}^2 \phi$ is the vorticity, $\{x, y\}$ represents the Poisson bracket operator in the perpendicular directions x and y, and the parallel direction aligns with z. $\alpha = \frac{T_0}{n_0 e \eta \omega_{ci}}$ scales the parallel dynamics in the model, which itself is determined by the electron response. Here $\omega_{ci} = Z_i q_i B / m_i$ is the ion cyclotron frequency and η is the well known Spitzer electron resistivity [10]. $\kappa = \frac{\rho_{s0}}{\lambda_q}$ determines the radial density scale length, where $\rho_{s0} = \frac{\sqrt{m_i T_e}}{eB}$ is the hybrid Larmor radius and $\lambda_q = \frac{1}{n_0} \frac{\partial N}{\partial x}$, which is the heat flux width.

Note that this set of equations adds new physics to previous 3-D implementations of the Hasegawa-Wakatani equations in that previously there was no parallel diffusion term, just the term $\alpha(n - \phi)$. With parallel diffusion switched on, these equations embody one more logical step towards a full plasma model since it enables us to study the effects of parallel-diffusion on the code's stability. It was found that the parallel-diffusion term has a stabilising effect on the numerics, as expected.

Listing 2.5.2 shows the physics- and meta-parameters of the proof-of-principle simulation, results are plotted in Figures 4 and 5 for the case without recombination and with recombination respectively. Recombination is switched off by setting
$num\_recombination\_particles\_per\_cell\_per\_step = 0$ since the physical rates are determined from the plasma state and not configured by the user.

Referring to the upper-right panel of Figs. 4 and 5, both showing the snapshot in time of the electron density at the late non-linear stage of the simulation, it can be seen that the structures are more diffuse without recombination physics. One lesson to take away from this is to ensure that the spatial distribution of the electron density remains smooth through choosing a suitably large number of neutral particles considering that $1/\sqrt{N}$ particle noise can sometimes cause large oscillations in the higher order basis functions. A subject of future work is on the topic of efficient use of particles such as importance sampling, splitting and roulette and low discrepancy numbers to ensure that this problem is mitigated on multiple fronts. These techniques will allow us to retain the good signal to noise ratios and alleviate some of the associated computational cost.

Listing 10: The recombination `ParticleLoop`.

```
inline void recombination_post_evaluate_fields(const double dt, const int num_added_recomb_particles) {
  const double k_dt = dt;
  const double k_dt_SI = dt * m_t_to_SI;
  //const double rate = 1e-19; // this is a guess! // ionisation rate is 1.02341e-14
  const double temperature_eV = m_TeV; // in general this would be a fluid quantity on a ParticleDat
  const double k_n_scale = 1 / m_n_to_SI;
  const double k_particle_pseudo_volume = m_particle_region_volume / num_added_recomb_particles;

  // Perform a position update style kernel on particles with even values of
  // ID[0].
  auto loop = particle_loop(
    "recombination_particle_loop",
    m_particle_group,
    [=](auto ELECTRON_DENSITY, auto PARTICLE_ID, auto COMPUTATIONAL_WEIGHT, auto SOURCE_DENSITY){
      if  (PARTICLE_ID.at(0) < 0) {
        const auto n_SI = ELECTRON_DENSITY.at(0);
        REAL rate = recombination_rate(temperature_eV); //m_recombination_rate; // ionisation rate is 1.023
        REAL weight = rate * k_dt_SI * n_SI * n_SI * k_particle_pseudo_volume;
        COMPUTATIONAL_WEIGHT.at(0) = weight; // neutral weight
        PARTICLE_ID.at(0) *= -1; // no longer negative
        SOURCE_DENSITY.at(0) -= weight * k_n_scale / k_dt; // plasma loses mass
      }
    },
    Access::read(Sym<REAL>("ELECTRON_DENSITY")),
    Access::write(Sym<INT>("PARTICLE_ID")),
    Access::write(Sym<REAL>("COMPUTATIONAL_WEIGHT")),
    Access::write(Sym<REAL>("SOURCE_DENSITY"))
  );

  loop->execute();

  return;
}
```

Variable	**Value**	**Description**
`TimeStep`	0.000625	Time per iteration in simulation units
`NumSteps`	64000	Total number of iterations
`TFinal`	`NumSteps`$\times$`TimeStep`	Final time in simulation units
`IO_InfoSteps`	`NumSteps`/1600	Time per info dump
`IO_CheckSteps`	`NumSteps`/160	Time for check file dump
`Bxy`	1.0	Magnetic field strength
`d22`	0.0	Coeff for Helmholtz solve
`HW_alpha`	0.1	HW alpha parameter
`HW_kappa`	3.5	HW alpha parameter
`HW_omega_ce`	0.1	electron cyclotron frequency
`HW_nu_ei`	1.0	HW parameter
`s`	0.5	Scaling factor for ICs
`num_particles_per_cell` i	-1	Number of neutral particles per cell
`num_particle_steps_per_fluid_step`	1	Number of substeps of neutral algorithm per fluid advance
`num_particles_total`	100000	Total number of particles at initialisation
`particle_num_write_particle_steps`	`IO_CheckSteps`	Particle data written every nth step
`particle_number_density`	10^{16}	The initial background number density of the neutrals
`particle_position_seed`	1	Seed for random number generator
`particle_thermal_velocity`	1.0	The thermal speed in simulation units for the initial particle set
`particle_drift_velocity`	2.0	The drift speed in simulation units for the initial particle set
`particle_source_width`	0.2	The spatial source width in simulation units for the initial particle set
`Te_eV`	10	Temperature in eV used to compute ionisation rate
`n_bg_SI`	10^{18}	Assumed background density in SI [m^{-3}]
`t_to_SI`	2×10^{-4}	Time normalisation factor in [s]
`n_to_SI`	1×10^{17}	Number density normalising factor in [m^{-3}]
`growth_rates_recording_step`	1	A diagnostics timestep for energy and enstrophy
`num_recombination_particles_per_cell_per_step`	100	... for the recombination process

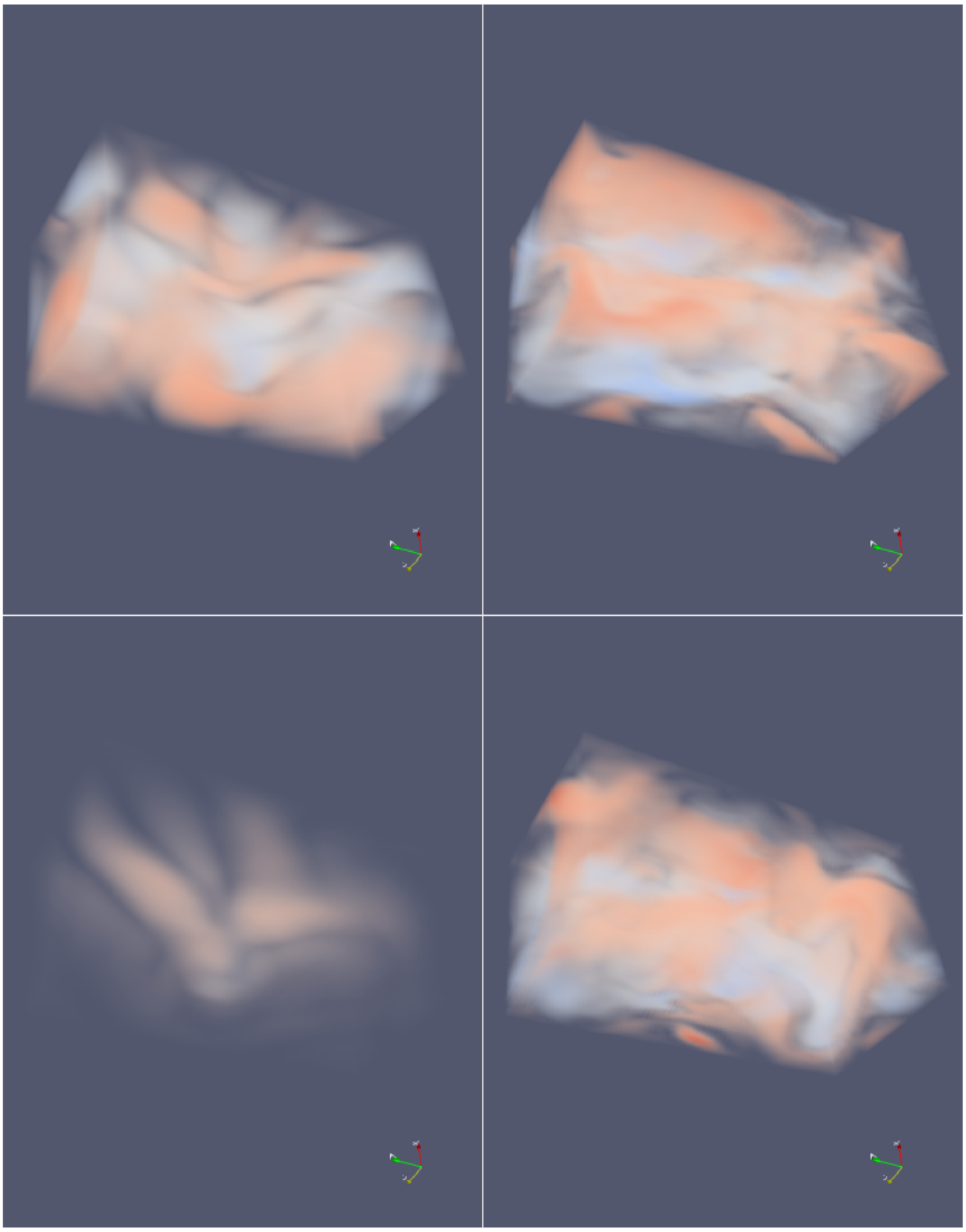


Figure 4: Evolution of the electron density evolving according to the 3-D Hasegawa-Wakatani equations in the presence of ionisation without recombination. The four quadrants depict four stages in the simulation: lower-left, early linear; top-left, late linear; lower-right, early non-linear; top-right, late non-linear.

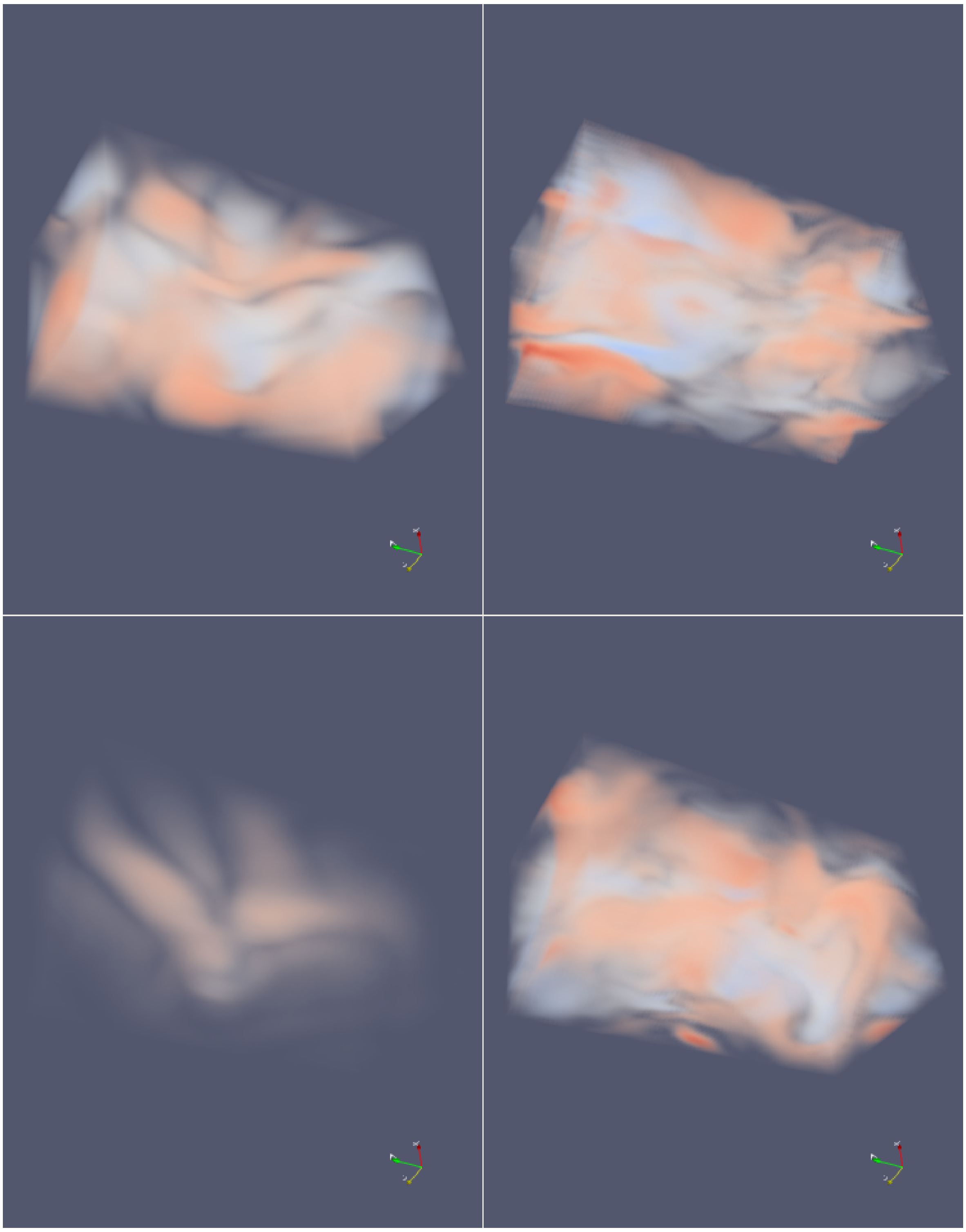


Figure 5: As in 4 except with the addition of recombination physics. Note that recombination alters the non-linear evolution of the simulation.

3 Summary

In this report we describe further developments made to the NESO framework enabling the parallel dynamics in a 3-D Hasegawa-Wakatani system coupled to neutral particles involving ionisation and recombination physics.

Significant effort has gone into augmenting the capability of the NESO-Particles framework so that the user may write performance portable GPU-capable particle kernels completely without knowledge of SYCL, the performance portability framework used within NESO-Particles allowing it to run on multiple architectures in a vendor agnostic way. Importantly, this abstracts SYCL from NESO such that other performance portable frameworks could be used to enable execution of particle kernels on GPUs without any breaking changes to NESO itself. This is foundational to robust software that respects the principle of separation of concerns.

Deploying the latest changes made in NESO-Particles to NESO in the form of new recombination physics, we have been able to observe fundamental changes to the nonlinear behaviour of the turbulent plasma state. This capability is a prerequisite to accurate modelling of the exhaust region in tokamaks and hence their design.

Acknowledgement

The support of the UK Meteorological Office and Strategic Priorities Fund is acknowledged.
This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk).

References

[1] William Robert Saunders, James Grant, and Eike Hermann Müller. A domain specific language for performance portable molecular dynamics algorithms. *Computer Physics Communications*, 224:119–135, 2018.

[2] W. Saunders, J. Cook, and W. Arter. 2-D Model of Neutral Gas and Impurities. Technical Report CD/EXCALIBUR-FMS/0061-M4.2, UKAEA, 12 2021. https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/ukaea_reports/CD-EXCALIBUR-FMS0061-M4.2.pdf.

[3] W. Saunders, J. Cook, and W. Arter. High-dimensional Models Complementary Actions 2. Technical Report CD/EXCALIBUR-FMS/0062-M4.3, UKAEA, 3 2022. https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/ukaea_reports/CD-EXCALIBUR-FMS0062-M4.3.pdf.

[4] S. Wright, E. Higgins, G. Mudalige, Z. Lantra, B.F. McMillan, and T. Goffrey. Proposal for a Particle DSL. Technical Report 2057699-TN-04-2, UKAEA Project Neptune, 2023. https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/2057699/TN-04-2.pdf.

[5] S. Wright, E. Higgins, C. Ridgers, G. Mudalige, and Z. Lantra. DSL Mini-application Performance Report Revision 1.0. Technical Report 2067270-TN-05-1, UKAEA Project Neptune, 2024. `https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/2067270/TN-05-1.pdf`.

[6] 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, 2012.

[7] J. Cook, W. Saunders, and W. Arter. 1-D and 2-D particle models. Technical Report CD/EXCALIBUR-FMS/0070, UKAEA Project Neptune, 2022.

[8] R. K. Janev, Ulrich Samm, and Detlev Reiter. Collision processes in low-temperature hydrogen plasmas. 2003.

[9] J. Cook, W. Saunders, O. Parry, M. Barton, and W. Arter. 3-D integrated particle and continuum model. Technical Report CD/EXCALIBUR-FMS/0073, UKAEA Project Neptune, 2023.

[10] Lyman Spitzer and Richard Härm. Transport Phenomena in a Completely Ionized Gas. *Physical Review*, 89(5):977–981, March 1953.