UK Atomic Energy Authority

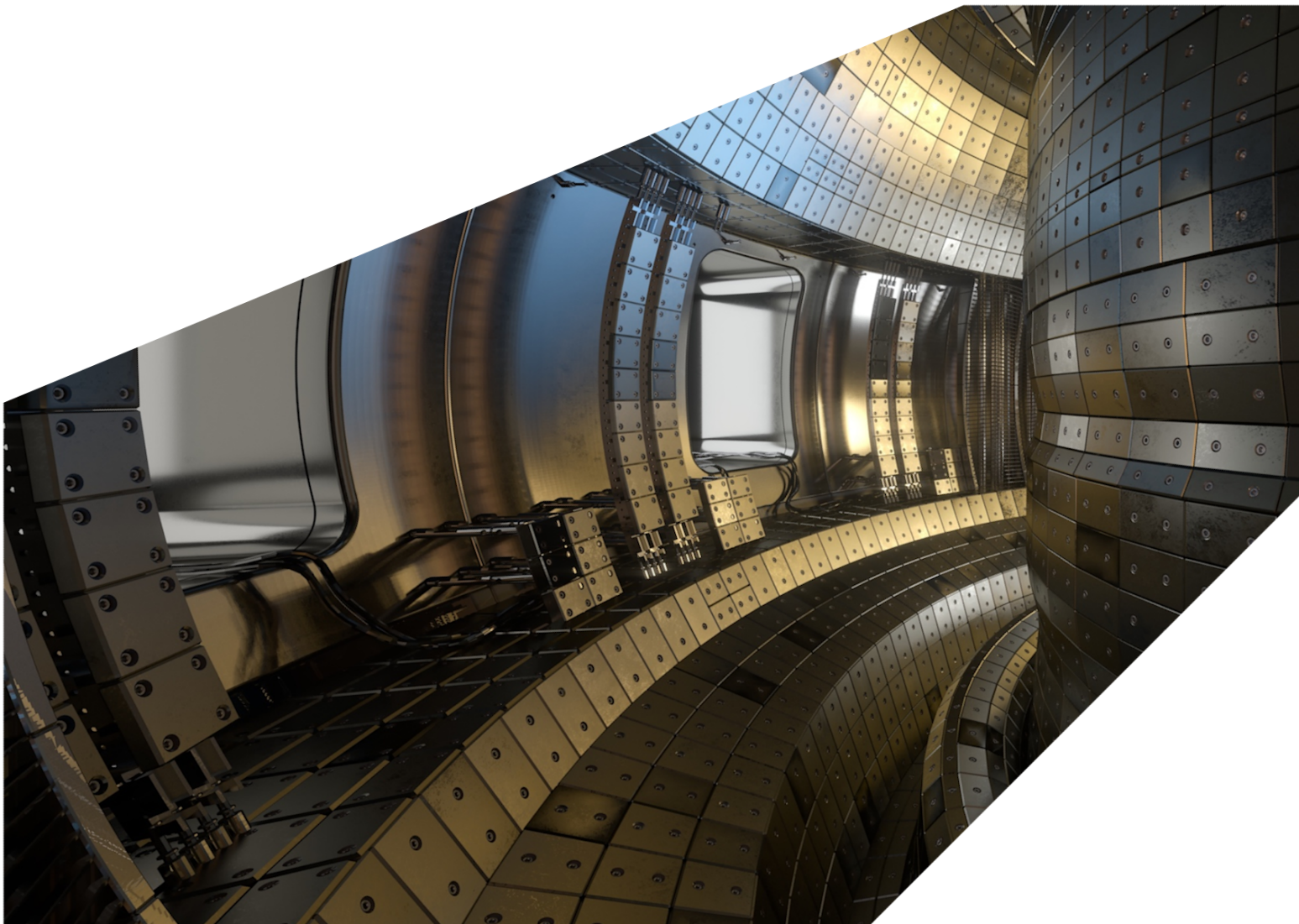
ExCALIBUR

2-D Model of Neutral Gas and Impurities

M4.2

Abstract

The report describes work for ExCALIBUR project NEPTUNE at Milestone M4.2. The task of representing neutral gas and impurities is a subproblem of the more general task of implementing particle descriptions and algorithms. In this report we discuss a DSL and implementation for describing particle-based algorithms that is performance portable. This DSL aims to be sufficient for both the particle component of a PIC algorithm and the description of neutral species represented by particles.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:	
	UKAEA Reference:	CD/EXCALIBUR-FMS/0061
	Issue:	1.00
	Date:	December 17, 2021
Project Name: ExCALIBUR Fusion Modelling System		

	Name and Department	Signature	Date
Prepared By:	Will Saunders James Cook Wayne Arter BD	N/A N/A N/A	December 17, 2021 December 17, 2021 December 17, 2021
Reviewed By:	Rob Akers Advanced Computing Dept. Manager		December 17, 2021

1 Introduction

1.1 Overview

The NEPTUNE software is expected to model both plasma and neutral species. The distribution functions for these species may not be well approximated by Maxwell-Boltzmann distributions in critical regions of an edge simulation. In these regions a "kinetic" description is required. We consider kinetic representations formed as a collection of individual particles and in particular the implementation of these particle-based representations. These particles transport quantities of interest, such as charge and mass, throughout the simulation domain. In a Particle-in-Cell (PIC) simulation the motion of charged particles is tightly coupled to the evolution of electric and magnetic fields which are represented by a finite element or finite difference approximation.

Efficient implementation of the PIC algorithm requires a tight coupling between the particles and the continuum implementations of the fields. Furthermore efficient implementation of both particle- and finite element-based algorithms is architecture dependent. By implementing a separation of concerns approach the description of particle-based algorithms is separated from the actual implementation which performs the operations. The separation allows an algorithm to be written independently of the runtime hardware which allows an algorithm to be written with minimal knowledge of the hardware. Secondly the underlying framework may be extended and optimised for new hardware types without modifying or reimplementing algorithms written by users at the higher level.

A Domain Specific Language (DSL) provides an abstraction layer that performs the actual separation of concerns. We describe a proof-of-concept DSL based on an existing DSL for performance portable atomistic simulations. This DSL is embedded in Julia programming language such that we may evaluate our relatively new language in the context of the NEPTUNE use case. The Julia language is designed in particular for use in numerical computing and aims to exist as a language which is both high level and performant.

We target CPU and GPU hardware in a portable manner by utilising packages from the Julia ecosystem that provide abstractions for parallel looping operations at a higher level than the vendor provided API. Furthermore we implement distributed memory parallelism using MPI such that this Julia implementation will compose with existing MPI applications and libraries.

2 Representations of Distribution Functions

Plasma and neutral species are both described by a time dependent distribution function $f_s(\vec{r}, \vec{v}, t)$ where $\vec{r}$ is a point in space, $\vec{v}$ is a point in velocity space, t is time and s is the species this distribution function refers to. The direct sum of the space of positions and the space of velocities is called the phase space and this distribution function describes the density of a particular species at each point in the phase space.

If we assume that a particular species is well described as an ideal gas and if we assume that within a region of space that the species is at thermal equilibrium then it is known that the velocity distribution of the species is a Maxwell-Boltzmann distribution. If it is known that a Maxwell-Boltzmann distribution is a good approximation for the plasma or neutral species in a region of space then it may be appropriate to model that particular species in that region of space as a fluid. In a fluid representation of a species s the distribution function $f_s(\vec{r}, t)$ is only a function of position in space and time. The equations of state that describe the time evolution of the system also make the assumption that a fluid description is appropriate.

If the velocity component of the distribution function is not well described by a Maxwell-Boltzmann distribution then a fluid description may well be insufficient to describe the behaviour of the species. In this scenario the representation of the distribution function must be extended to not only be a function of a point in physical space but also a point in velocity space. In this case the representation is referred to as "kinetic". Representing functions defined over a physical space is typically easier than a velocity space due to the constraints imposed by the particular problem of interest. Typically the physical space a simulation is concerned with is a finite and often fixed region of space that can be decomposed into cells. The union of these cells forms a mesh and meshes of this form are applied in finite element simulations. The finite element method (FEM) defines a set of basis functions in each cell which are subsequently used to approximate quantities of interest. The accuracy of these approximations is governed by the size of the cells itself and the polynomial order of the basis functions.

Representation of a function defined on a velocity space is less obvious. Like physical space, the bounds on the domain are well defined as nothing can travel faster than the speed of light c in any direction. However it is unrealistic to mesh this domain in a similar manner as physical space and thus techniques that represent the velocity space via a mesh must take particular care to discretise the space in a manner that is both accurate and computationally sensible. The focus of this report is an alternative approach where "particles" are used to provide the kinetic description. In this report a particle is a discrete object that exists at a point in physical space, i.e. particle i exists at a point $\vec{r}_i \in \Omega$ for some simulation domain Ω that describes the region of physical space where particles exist.

Each particle stores properties, in addition to position, that are used approximate quantities of interest. For example, particles that represent constituent species of the plasma would carry a quantity that holds the net charge of the particle. Furthermore, neutral species could be expected to have properties that indicated the species type. All moving particles would carry a property that represents the velocity of that particle and these particle velocities allow the particle description to provide a kinetic representation. Although the particle exists at a point in space, particles may have properties that describe shape and orientation. In this report we shall assume that particles

are Dirac deltas in both physical and velocity space. With this shape function the distribution f_s is approximated as

$$f_s(\vec{r}, \vec{v}) \approx \sum_{i=1}^{N} \delta(\vec{r} - \vec{r}_i)\delta(\vec{v} - \vec{v}_i) \tag{1}$$

where N is the number of particles and $\vec{v}_i$ is the velocity of particle i. In general particle shape is not restricted to Dirac deltas and alternatives such as B-splines exist. As the RHS of Equation (1) has empty support quantities are estimated at point in space by averaging over particles within a ball near the point of interest. For example, the approximation of a quantity q stored on particles would be approximated by

$$q(\vec{r}) = \int_{\vec{v}} q(\vec{r}, \vec{v}) f_s(\vec{r}, \vec{v}) d\vec{v} \approx \frac{1}{N_{B(\vec{r},\epsilon)}} \sum_{i \in B(\vec{r},\epsilon)} q_i \tag{2}$$

where $B(\vec{r}, \epsilon)$ is a ball of radius ϵ around point $\vec{r}$, $N_{B(\vec{r},\epsilon)}$ is the number of particles within this ball and q_i is the quantity q stored on particle i. As the approximation is computed as an average over particles the computation of a good approximation requires a statistically significant particle count at the evaluation point. Other weighting functions than the uniform one represented by a simple sum in Equation (2) may be needed, eg. functions that are biased towards the location of $\vec{r}$.

3 Domain Specific Language

Domain Specific Languages (DSLs) allow description of algorithms and processes within a language specifically designed for a particular domain. Complex implementations often involve a hierarchy of DSLs that ultimately abstract low-level implementation details from the high-level description of a simulation. In this report we discuss a DSL for the description of particle-based data and operations.

This high-level particle description is required to be able to describe physical processes that involve both charged and neutral species. Furthermore the NEPTUNE project could employ a particle in cell (PIC) technique to model the evolution of the plasma. As the name indicates, this approach requires specification of particle data and operations to manipulate particle data. We now outline a DSL for particle-based operations designed to meet the requirements of neutral species modelling and PIC algorithms.

3.1 Particle DSL

We describe an initial abstraction and a DSL inspired by and based on the PPMD framework [1]. The PPMD abstraction is suitable for describing groups of particles with per-particle properties. Global data can also be described within the abstraction. The abstraction defines looping operations, originally designed for molecular dynamics, that loop over particles and pairs of particles. This abstraction creates a separation of concerns where particle-based data structures and looping operations are described independently of the hardware that performs the computation or

stores the data. In the original PPMD implementation the abstraction is realised as a DSL embedded in the Python programming language. This Python implementation uses a code generation approach to produce C/C++ code for CPUs and GPUs at runtime for the particular operations defined by the user. A code generation approach is a method to achieve performance portability which is an important property of a successful NEPTUNE implementation.

A successful separation of concerns for particles allows a domain specialist such as a plasma physicist or engineer to describe particle-based operations without knowledge of the underlying hardware. Furthermore the execution hardware can be utilised without a reimplementation of the described algorithms. At the lower level optimisations and improvements can be investigated and implemented that subsequently benefit all operations described using the DSL without rewriting the higher level code.

As the original PPMD design was focussed on the MD use case we investigate modifications and extensions to the abstraction and implementation that are more suited to the plasma modelling use case. In particular a DSL for particles in NEPTUNE should be suited to describe both the charged species within a PIC context and the neutral species that are important to capture the correct physics in the tokamak edge plasma.

In recent years the Julia programming language [2] has been proposed as a solution to the two-language problem in numerical computing. As Julia is a attractive alternative to Python for our use case, we use this opportunity to embed our PPMD-inspired DSL within Julia for evaluation purposes. The Julia JIT compilation approach allows code generation to occur at runtime in a similar manner to the original PPMD approach. Furthermore the Julia package ecosystem includes packages that target accelerator style hardware such as GPUs directly from Julia. Composition with more traditional packages written in C/C++ using MPI is readily achievable as Julia supports calling C functions directly and a functional MPI package exists within the ecosystem.

For mesh-based representations, we assume that there will exist a DSL that allows a user to interface with an FEM library. In particular we assume that this DSL allows a function represented by particle data to be approximated by a function in a suitable finite element function space. We also assume that this interface allows a user to evaluate finite element functions at arbitrary locations. In particular a user will wish to evaluate functions at particle locations.

3.2 Data Structures

We now describe a set of data structures designed to capture particle data and auxiliary objects such as domains and global data. The performance of the PIC implementation is expected to be dependent on how tight the coupling is between the particle and FEM frameworks. The parallel scaling efficiency of an implementation is given by Amdahl's law [3] which states that the maximum possible parallel scaling is determined by the proportion of runtime that is sequentially executed. In modern HPC facilities the time taken to communicate data between compute nodes, for both arithmetic and bookkeeping purposes, is a significant contributor to this sequential runtime.

A technique to reduce inter-node communication is to employ parallel decomposition techniques that take into consideration both the location of data and where that data is required for computation. It is desirable to exploit the structure of the underlying computational problem to achieve

such a decomposition. In the case of PIC methods we know a priori that data from particles in a given mesh cell will be required by the FEM implementation to modify or evaluate FEM fields in the same cell. Hence to maximise data locality we design an approach that uses the same mesh and domain decomposition as the FEM implementation.

Efficient use of accelerator devices, such as GPUs, is sensitive to the time taken to prepare the data required for a loop. For example, in a typical heterogeneous architecture the host system memory is distinct from accelerator memory and explicit or implicit transfer of data must occur before a kernel launch. If the frequency and timing of this transfer is not carefully managed, either by the programmer or underlying implementation, the efficiency gains of using the accelerator device can be significantly damaged. In our initial implementation we allow the user to specify which device owns the particle data and assume that subsequent computation that uses this data is performed on the same device. Our interface does allow an extension where the computation device is not the same as the host device for the data.

3.2.1 Domain and Mesh

We assume that the computational domain is completely covered by a mesh and that this mesh is potentially a high-order mesh. We also assume that the mesh was generated and decomposed independently of the particle framework. Our initial implementation includes basic cuboid domains with periodic boundary conditions for testing purposes. The spatial domain decomposition approach partitions the mesh into contiguous regions that are uniquely owned by an MPI rank.

In our implementation the MPI rank that owns a partition of the mesh also owns the particles that reside in that mesh partition. We assume that the locally owned region of the mesh is surrounded by a halo region of cells that are copy of mesh cells owned by neighbouring MPI ranks. When a particle leaves the owned region the particle framework inspects the mesh halo to determine the new owning rank of the particle. To implement this transfer of functionality the particle framework must be able to map a position in space to an owning cell and map mesh cells to owning MPI ranks.

The mesh requirements of a PIC implementation are more involved than a traditional MD code where typically the computational domain is a cuboid (potentially a cuboid undergoing a linear transform to represent a skewed geometry). In MD a structured mesh is typically applied along with a parallel decomposition that maps directly onto an MPI Cartesian communicator. This structured approach greatly reduces the bookkeeping overhead required to transfer particles between MPI ranks as a point in space can be easily mapped to an MPI rank.

The boundary conditions of a PIC code suitable for NEPTUNE are significantly more involved than a MD code where typically periodic boundary conditions are applied. For PIC, the computational domain and mesh must include descriptions of the regions of space that act as sources and sinks for different particle species. For example, on the core facing boundary particles representing plasma species could be created. On the outer wall facing boundary plasma species may be converted to neutral species due to plasma-wall interactions.

There may also be volume, line and even point sources, possibly time-varying and controlled by feedback from other outputs of the code. The particle framework must allow a user to describe

both how these processes occur and where in the domain they occur.

The simplest domain we consider is a hypercuboid in one to three dimensions. We apply periodic boundary conditions to this domain. In Listing 1 we illustrate how to import the `PPMD` Julia package and create a cuboid domain. This domain is automatically decomposed across all MPI ranks on the communicator. We anticipate that in future the domain would be defined as the mesh that is used for the FEM implementation.

Listing 1: Create a cuboid domain with periodic boundary conditions.

```
# Import the PPMD package
using PPMD
# Define the extents in x,y and z of a cuboid domain.
extents = (4.0, 2.0, 3.0)
# Define the boundary condition.
boundary_condition = FullyPeriodicBoundary()
# Create the domain, default MPI_COMM_WORLD.
domain = StructuredCartesianDomain(boundary_condition, extents)
```

3.2.2 Particle Data

As in the original PPMD DSL, per-particle properties are specified with the `ParticleDat` data structure. A `ParticleDat` is initialised with the number of components and a data type that will be created and stored per particle. In atomistic simulations, particularly when using domain decomposition, it is important to know exactly where each particle is in the simulation domain. To indicate the particle positions to the underlying implementation the `ParticleDat` which stores the particle positions is explicitly indicated.

A set of particles is defined as a collection of `ParticleDats` and is defined with a `ParticleGroup` object. The `ParticleGroup` object combines one or more `ParticleDats` with a domain, such as the domain described in Section 3.2.1, and a target device. The target device indicates the primary storage location for the particle data which may be a CPU or GPU. In this initial implementation we use the Julia array abstraction to create CPU and GPU arrays. In Listing 2 we illustrate how to create a `ParticleGroup` with multiple `ParticleDat` instances.

Listing 2: Create a group of particles in a domain with the same per-particle properties. All particle data is stored on the target device.

```
target_device = KACUDADevice()
A = ParticleGroup(
    # The domain containing these particles.
    domain,
    # The per-particle properties.
    Dict(
        # Create a property "P" of 3 components per particle (Float64)
        # and indicate these 3 components are positions.
        "P" => ParticleDat(3, position=true),
        # 3 Float64 components per particle labelled "B".
```

```
        "B" => ParticleDat(3),
        # 1 Int64 per particle labelled "C".
        "C" => ParticleDat(1, Int64),
    ),
    # The device memory where the particle data is primarily stored.
    target_device
)
```

3.2.3 Global Data

Typically scientific algorithms require global data in addition to particle data. Global data presents additional challenges for a parallel framework as a parallel implementation must ensure that the data access is correct and efficient. Reading from constant global data typically is straightforward in comparison to writing to global data where write contention must be carefully considered.

For example, a user could wish to accumulate the kinetic energy of a set of particles which are globally distributed. They may also wish to assemble array-based structures such as a histogram or matrix. For this particle framework we shall assume that access to global data is either read only or a write access involving a commutative operator. We provide the `GobalArray` data structure, as illustrated in Listing 3, which represents global data consistently across MPI ranks.

Listing 3: Create a `GlobalArray` to hold elements that are acessible in looping operations and in Julia.

```
target_device = KACUDADevice()
kinetic_energy = GlobalArray(1, target_device)
```

As with particle data, the constructor of the `GlobalArray` describes the target device for the data. This indicates where the data will be required which in turn determines the algorithm that should be applied to access the data. For example, the implementation could use a different pattern to perform elementwise reduction operations on CPUs than on GPUs. These architecture specific implementations are particularly important for efficiency on modern hardware.

3.2.4 Adding and Removing Particles

In a NEPTUNE simulation, particles will be created and destroyed through physical processes such as boundary conditions and interaction terms. Large particles may also be subdivided into smaller particles. PPMD provides methods `add_particles` and `remove_particles` that add particles to and remove particles from a `ParticleGroup`. In Listing 4 we demonstrate adding and removing particles. The `add_particles` function is collective on the passed `ParticleGroup` as this function assumes that the new particles may be located anywhere in the simulation domain. When an input particle is not owned by the MPI rank the particle was created on then communication must occur to transfer the particle to the correct MPI rank. A future extension, likely to be useful for the plasma use case, would be a function that allows users to create particles on the local domain such that global synchronisation is not required.

Listing 4: Add and remove particles from a `ParticleGroup`.

```
# Import the PPMD package and MPI
using PPMD
using MPI

# choose a target device
target_device = KACPU()

# Create a boundary condition and domain
extents = (1.0, 1.0)
boundary_condition = FullyPeriodicBoundary()
domain = StructuredCartesianDomain(boundary_condition, extents)

# Create a ParticleGroup with particle properties "P" for positions
# and "A" an arbitrary property.
A = ParticleGroup(
    domain,
    Dict(
         "P" => ParticleDat(2, position=true),
         "A" => ParticleDat(1),
    ),
    target_device
)

# On MPI rank 0 add 10 particles. The "A" property is not specified
# and will be initialised to 0.
if MPI.Comm_rank(domain.comm) == 0
    N = 10
    add_particles(
        A,
        Dict(
             "P" => rand_within_extents(N, domain.extent),
        )
    )
else
    add_particles(A)
end

# On each MPI rank remove the first particle from the ParticleGroup.
if A.npart_local > 0
    remove_particles(A, [1,])
end
```

3.3 Looping Operations

We have described data structures for particle and global data in a flexible and portable manner. Efficiently operating on this data requires looping operations which are performance portable. The original PPMD DSL included operations for looping over particles and looping over pairs of particles. The pairwise particle operation in the original PPMD is designed for inter-particle interactions such as the inter-atomic potentials in MD simulations. Such interactions, commonly involving neutral species, will require treatment by NEPTUNE software, as will interactions with a background density of other species. However, we currently do not consider operations that can be expressed in a pairwise manner and focus on a loop over particles.

3.3.1 Particle Loop

As alluded to, a `ParticleLoop` is a loop over all particles in a `ParticleGroup`. The loop can read and write to the data stored on each particle in `ParticleDats` and can read or modify global data stored in `GlobalArrays`. A `ParticleLoop` is constructed with a kernel and a set of access descriptors. The kernel describes the particular operation that this loop will perform for each particle and takes the form of a Julia string literal that contains the Julia source code for the kernel.

The access descriptors map the symbols in the source code for the kernel to the matching data structures. Crucially the access descriptors describe exactly how the kernel accesses the data. The implementation inspects the access descriptors and uses the information to generate efficient code at runtime for the access type. This information is required by the framework to correctly and efficiently execute the `ParticleLoop` on the target hardware. In Listing 5 we illustrate how a user would create and execute a `ParticleLoop` that operations on both particle and global data.

Listing 5: Create and execute a `ParticleLoop`.

```
# Standard initialisation that creates target device and domain
using PPMD
using MPI
# target_device could also be KACUDADevice()
target_device = KACPU()
extents = (1.0, 1.0)
boundary_condition = FullyPeriodicBoundary()
domain = StructuredCartesianDomain(boundary_condition, extents)

# Create a ParticleGroup with positions "P" and velocities "V".
A = ParticleGroup(
    domain,
    Dict(
        "P" => ParticleDat(2, position=true),
        "V" => ParticleDat(2),
    ),
```

```
    target_device
)

# Add particles to the domain with random positions and velocities.
if MPI.Comm_rank(domain.comm) == 0
    N = 10
    add_particles(
        A,
        Dict(
            "P" => rand_within_extents(N, domain.extent),
            "V" => rand(Float64, N, 2),
        )
    )
else
    add_particles(A)
end

# Create global storage to store the kinetic energy.
kinetic_energy = GlobalArray(1, target_device)

# Create a kernel for a ParticleLoop that
# 1) Modifies the position of the particle
# 2) Increments the kinetic energy with the contribution from this
#    particle.

# Runtime constants can be placed into the kernel as literals.
dt = 0.01
move_kernel = Kernel(
    # Arbitrary name to identify this kernel in say profiling.
    "move_kernel",
    # Kernel code
    """
    # Access the first and second components of positions and
    # velocities
    P[1] += $dt * V[1]
    P[2] += $dt * V[2]

    # Increment the GlobalArray storing kinetic energy.
    KE[1] += V[1] * V[1] + V[2] * V[2]
    """
)

# Create the loop
move_loop = ParticleLoop(
    # Target device for execution, in theory this could be different
    # to the device storing the particles.
```

```
    target_device,
    # Our compute kernel defined above.
    move_kernel,
    # This dictionary maps from symbols in the kernels to Julia objects
    # and lists the access descriptor for each object.
    Dict(
        "P" => (A["P"], WRITE),
        "V" => (A["V"], READ),
        "KE" => (kinetic_energy, INC),
    )
)

# execute the ParticleLoop
execute(move_loop)

# print to stdout the computed kinetic energy. The reduction across
# MPI ranks is automatic.
@show kinetic_energy[1]
```

The example in Listing 5 demonstrates both the separation of concerns and portability of this approach. A user can target multiple hardware architectures at the DSL level by changing the target device. Furthermore the user can utilise distributed memory parallelism with minimal knowledge of MPI. In section 4 we give an overview of the underlying implementation and how this approach gives performance in addition to portability.

3.3.2 Potential Extensions

We currently describe a loop over particles that visits all particles within a `ParticleGroup` which is applicable to the looping operations we expect to find in the core of a PIC code. In addition to the core operations that evolve a PIC simulation it is expected that there will be auxiliary loops that do not need to visit each particle in the `ParticleGroup`. These partial loops will arise in the case of user defined diagnostic loops that compute quantities of interest from a subset of the global set of particles.

For example, consider a diagnostic that computes the flux of particles through a surface. We assume that the user holds an algorithm that can identify which side of the surface a given particle exists on. Hence the user can identify when a particle has passed through the surface by storing the previous side index on each particle in a `ParticleDat` and comparing at each time step. In principle the `ParticleLoop` we describe in Section 3.3.1 is sufficient for finding all particles that passed through the surface in the last time step as all particles are visited by the loop. However the surface may cover a relatively small portion of the simulation domain and hence visiting all particles in the simulation domain would be inefficient.

A more efficient approach would only loop over particles that were close to the surface. If we assume the user holds a method to identify particles close to the surface for a certain number of time steps then this method can be applied to reduce the number of visited particles for the

diagnostic loop. The ability to reuse the set of identified particles would be crucial to amortise the setup cost and provide an increase in efficiency.

4 Implementation

The implementation is required to store data and execute user defined loops on the target device requested by the user. The Julia language provides an array type that stores data on a host system in a manner that is familiar to users of MATLAB and NumPy. This array type is typically extended by the packages which provide support for accelerator devices. For example, the standard Julia array type is `Array` and the CUDA array type is `CuArray` and a user may use a `CuArray` in a highly similar manner to the host array type with the exception that the CUDA variant is stored in device memory. Using this array abstraction the implementation selects the target device that should store an array by simply choosing the array type that corresponds to that device. Hence this array abstraction allows the particle implementation to choose where memory is allocated in a manner that is transparent to the user.

One of the main motivations to investigate Julia for the NEPTUNE project is the capability of Julia to solve the two-language problem. The two-language problem refers to the classical requirement for a high-level user facing language to be used in conjunction with a low-level language for performance. Julia employs a type analysis and JIT compilation approach to translate user written code into optimised machine code at runtime. Portability to accelerator style compute devices, for example GPUs, is performed through the Julia packages provided by the hardware vendors and contributors. For example, a user could use the Julia CUDA package to implement CUDA kernel that execute on the device without touching CUDA C/C++.

From PPMD developer perspective there is a reasonably flexible set of approaches that could be employed. The abstraction and corresponding DSL form a separation of concerns between a user that describes data structures and looping operations and how the underlying framework actually provides the storage and looping implementation. The framework could implement a custom set of looping implementations per architecture. This per architecture approach would be computationally most efficient but would require loop implementations per architecture. Hence from the maintenance and implementation cost point of view it would be beneficial to restrict the number of looping types available in the DSL to minimise the per architecture implementation cost.

Clearly developers would prefer not to implement algorithms using a vendor specific API if a suitable alternative vendor agnostic approach is available. The `KernelAbstractions.jl`[4] Julia package can be thought of as the Julia analogue of SYCL and is a higher level interface than the vendor specific APIs. The abstraction provided by `KernelAbstractions.jl` follows the now typical pattern of defining a parallel loop as the combination of a compute kernel and an iteration set. We use this `KernelAbstractions.jl` package to target CPU and GPU architectures directly from Julia. It should be noted that like SYCL, `KernelAbstractions.jl` does not solve the problem that an algorithm efficient on one architecture may well not be efficient on another. Reduction type operations are an example of a common operation which requires an architecture specific implementation to be efficient.

In the proof of concept Julia implementation of PPMD we implement the `ParticleLoop` operation

by generating secondary Julia source code that uses the `KernelAbstractions.jl` package. This approach allows the implementation to exploit the JIT compilation offered by Julia at runtime whist targeting two architectures directly from Julia. Alternatively the framework could generate C++ code using SYCL or Kokkos to provide an implementation for the `ParticleLoop` described in the DSL.

5 Summary

Particle methods are a typical approach to the modelling of neutral and plasma species with a kinetic description. Efficient implementation of particle methods is highly non-trivial and requires specialist knowledge of the target hardware, hardware which is subject to change as technology developes. Often users wish to describe particle-based methods independently of the HPC hardware applied at runtime.

We described a DSL based on an existing abstraction for particle operations. This approach creates a separation of concerns that benefits both the domain specialist and computational scientist. We develop this DSL as an embedded DSL within the programming language Julia, which has recently emerged as a solution to the two-language problem. By using a combination of code generation and the `KernelAbstractions.jl` Julia package we target CPU and GPU architectures directly from Julia. This approach does not exclude a future implementation that generates SYCL/Kokkos code instead of Julia code.

In this report we describe the DSL for describing particle operations only. Future work should describe the transfer of data to and from finite element-based descriptions. These transfer operations are expected to be based around projection and evaluation and are essential for the implementation of a PIC algorithm.

Acknowledgement

The support of the UK Meteorological Office and Strategic Priorities Fund is acknowledged.

References

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

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483âĂŞ485, New York, NY, USA, 1967. Association for Computing Machinery.

[4] KernelAbstractions.jl. `https://github.com/JuliaGPU/KernelAbstractions.jl`, 2021. [Online; accessed 16-December-2021].