FFLUX Developer's Manual

Benjamin Symons

January 2022

1 Introduction

The purpose of this manual is to aid anyone that wishes to continue the development of the code DL_FFLUX. The focus will be on code that is specific to DL_FFLUX. Anyone wishing to learn more about the standard DL_POLY side of things is referred to the DL_POLY manual. If you see "IMPORTANT NOTE" then it generally refers to something that took me weeks or even months to figure out.

2 Architecture of DL_FFLUX

The FFLUX code is integrated with the code of DL_POLY 4.08 (with a couple of files from 4.09 with bug fixes). Recent work[1] has successfully integrated FFLUX into DL_POLY's domain decomposition MPI.

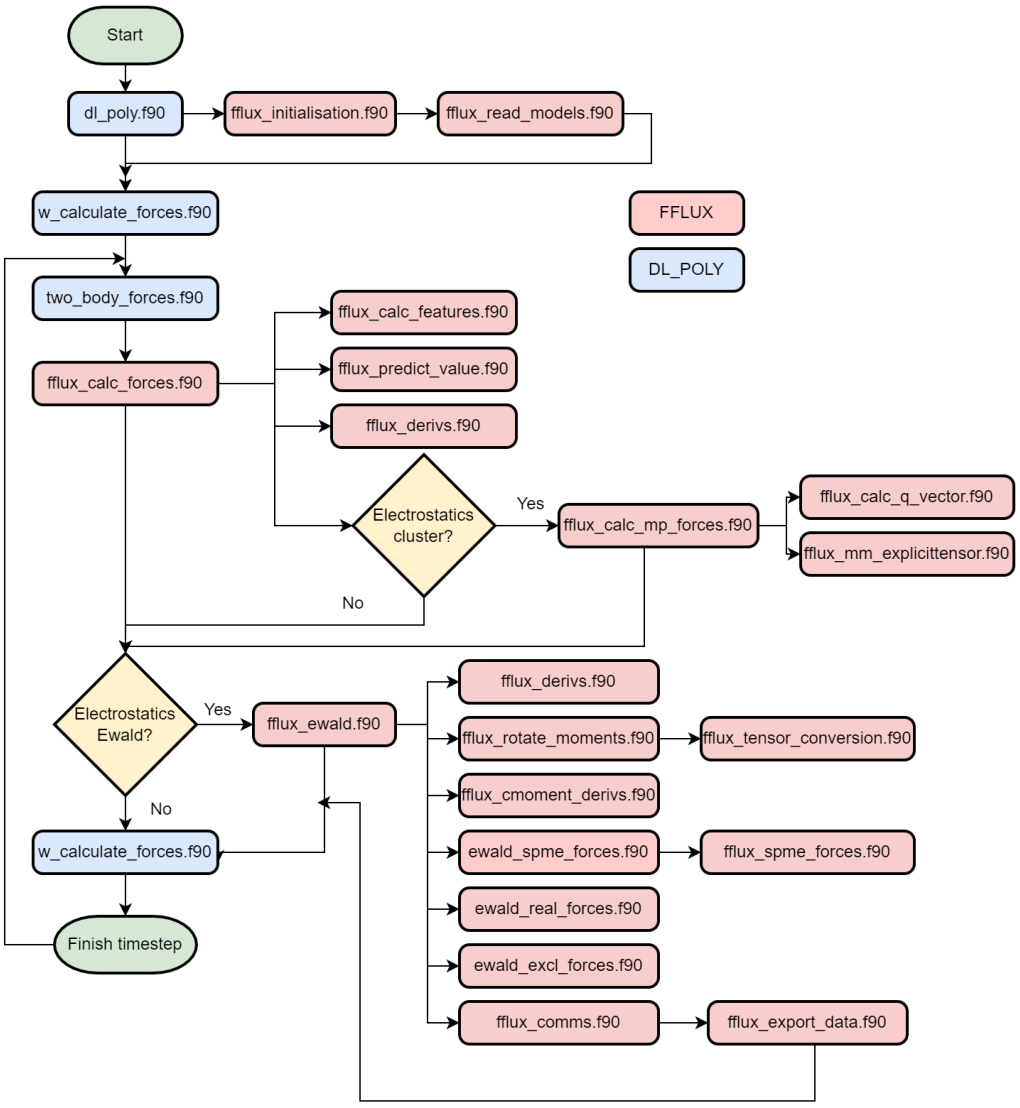

Figure 1: A flowchart demonstrating the overall architecture of DL_FFLUX. DL_POLY routines are shown in blue and FFLUX specific routines in red.

Figure 1 demonstrates the overall order and relation of routines although it is not exhaustive (the diagram would be to complex). The key parts that have been missed out are

to do with the initial one-off DL_POLY setup (e.g. reading input files) as well the starting and ending of timesteps. Almost all of this is done with standard DL_POLY code, some of which will be discussed in the parallelisation section. The routines in Figure 1 that begin "ewald_" were originally DL_POLY routines but have been heavily modified in order to compute extra force terms that arise from having multipole moments that depend explicitly on atomic positions.[2] As such, these routines are in reality a mixture of FFLUX specific and DL_POLY code. There are 3 sets of Ewald routines in DL_FFLUX (and DL_POLY). The first set are shown in Figure 1 and deal with point charge electrostatics only, the second set deal with interactions up to quadrupole-quadrupole and the files are called ewald_real_mforces_d.f90 etc. The first two sets implement explicit equations for the electrostatics. The final set are much slower and hand interactions up to hexadecapole-hexadecapole, these are called ewald_real_mforces.f90 and use recurrence relations.

3 Subroutines

This section will discuss each of the FFLUX subroutines in DL_FFLUX. At minimum some comments about the purpose of the routine as well as the reason why it has been done the way it has will be given. For more complex routines, more detail will be given.

3.1 fflux_module

This routine contains derived type definitions as well as several important mathematical functions.

```
! Derived data type for Kriging models - BCBS 5/20
  Type Model

    Character(Len=128) :: SystemName,AtomName
    Integer                        :: nTrain,nFeatures,nPredPerAtm,natms_model,nModels
    Integer,           Allocatable :: ALF(:),non_ALF(:)
    Real( Kind=wp),    Allocatable :: trained_value(:,:,:),weights_krig(:,:),mu(:), &
                                    & Theta(:,:),p_krig(:,:)
    !Procedure(KernelType), Pointer :: Kernel - Placeholder until KernelType
    !implemented.

  End Type Model

! Array of Model types - Size will be specified in KRIGING.txt
  Type ( Model ), Target,  Allocatable :: Models(:)

! Derived data type for pointers to the Model type - 5/20 BCBS
! Needed to construct an array of pointers to Model types
  Type ModelPtr

    Type ( Model ), Pointer :: ptr

  End Type ModelPtr
```

Figure 2: fflux_module Model type

The **Model** derived type definition is shown in Figure 2. For each unique Gaussian process regression (GPR) model that is found in the CONFIG file, an instance of the **Model** type is created. Each atom points to an instance of the **Model** type. An instance of the **Model** type for a given atom contains the models for all properties (IQA energy and multipole moments) i.e. each **Model** actually stores the data for all models for a given atom. The

ModelPtr derived type is necessary to create an array of pointers to an array of instances of the **Model** type.

Model%nTrain	Number of training points.
Model%nFeatures	Number of features.
Model%nPredPerAtm	Number of properties predicted (typically will be 1 IQA energy + 25 multipole moment components = 26).
Model%natms_model	Number of atoms in the model.
Model&ALF(:)	Array of size 3, stores the indices (as indexed in the model file) of the atoms that form the ALF.
Model%non_ALF(:)	Array of size **Model%natms_model**-3. Stores the indices (simply indexed 3,4,...,natms_model) of the remaining atoms that are not in the ALF but are still part of the model.
Model%trained_value(:,:,:)	Stores the x values (inputs) for each of the training points for the model. Size is (**nPredPerAtm**,**nTrain**,**nFeatures**).
Model%weights_krig(:,:)	Stores the weights for the model. Size is (**nPredPerAtm**,**nTrain**).
Model%Mu(:)	Stores the mean for the model. Size is (**nPredPerAtm**).
Model%Theta(:,:)	Stores the hyperparameters θ. Size is (**nPredPerAtm**,**nFeatures**).
Model%p_krig(:,:)	Currently un-used as p=2 is fixed for the RBF kernel, may be useful later.

3.2 fflux_initialisation

This routine is called once at the very start of a run and sets up several useful variables that persist across an entire run. It also calls fflux_read_models.

3.3 fflux_read_models

This routine is called once at the start of a run and reads in the model files located in the model_krig directory. The routine is written in such a way that it is simple to extend if more information is added to the model files. Each section in the model file is headed with "[SECTION]". If a new section is added, then a new if statement can be added to look for it and the **Model** type updated accordingly.

```
!Work out total number of models.
Do i=1,Size(model_names)
  Do j=1,Size(unique_model_names)
    If (model_names(i) .EQ. unique_model_names(j)) Then
      Exists = .TRUE.
    End If
    If (Exists) Exit
  End Do
  If (Exists .NEQV. .TRUE.) Then
    numKrigModels = numKrigModels + 1
    unique_model_names(numKrigModels) = model_names(i)
  End If
  Exists = .FALSE.
End Do
```

Figure 3: fflux_read_models code to determine number of unique models.

model_names(:)	This array is filled during the reading of the CONFIG file with the name of the GPR model that is associated with each atom.
unique_model_names(:)	The array **model_names(:)** is used to figure out how many unique models there actually are (shown in Figure 3) the names of these are stored in this variable so that FFLUX knows how many models to look for.
Models(:)	An array of instances of the **Model** derived type.
model_list(:)	An array of derived type **ModelPtr**. The array is **Natms** long and each element is a pointer to an element of **Models(:)** i.e. each atom points to an instance of the **Model** type which stores all of the information about the GPR model for that atom. Given that the same model is used for many atoms in a simulation this way of doing things is more efficient.

```
Do i=1,Size(model_list)
  globalAlf(i,:) = model_list(i)%ptr%ALF(:) + (i-model_list(i)%ptr%ALF(1))
  non_alf_atms(i,:) = model_list(i)%ptr%non_ALF(:) + (i-model_list(i)%ptr%ALF(1))
  atm_label_alias(i) = model_list(i)%ptr%ALF(1)
End Do
```

Figure 4: fflux_read_models code to genereate the globally indexed ALF and non-ALF arrays.

Figure 4 demonstrates the code that turns the ALF and non-ALF arrays from the model files and turns them into global arrays with entries for each atom. Indexing is important in DL_FFLUX (if MPI is switched on) and later, a locally indexed ALF must be generated. The array **atm_label_alias(:)** also preserves the relationship between the global index of an ALF and its index within the model.

3.4 fflux_calc_forces

This routine predicts the IQA energies and computes the resulting IQA forces. If the electrostatics is computed in cluster mode (direct Coulomb sum) then the relevant routine is called from here.

```
!Compute local ALF
counter = 1
mpi_list = 0
exists = .FALSE.
Do i=1,nlast
  alf(i,1) = i
  alf(i,2) = local_index(globalAlf(ltg(i),2), nlast,lsi,lsa)
  alf(i,3) = local_index(globalAlf(ltg(i),3), nlast,lsi,lsa)
  If (i ⩽ natms) Then
    If (alf(i,2) > natms) Then
      Do j=1,Size(mpi_list)
        If (alf(i,2) .EQ. mpi_list(j)) Then
          exists = .TRUE.
          exit
        Else If (mpi_list(j) .EQ. 0) Then
          exit
        End If
      End Do
      If (exists .EQV. .FALSE.) Then
        mpi_list(counter) = alf(i,2)
        counter = counter + 1
      End If
      exists = .FALSE.
    End If
    If (alf(i,3) > natms) Then
      Do j=1,Size(mpi_list)
        If (alf(i,3) .EQ. mpi_list(j)) Then
          exists = .TRUE.
          exit
        Else If (mpi_list(j) .EQ. 0) Then
          exit
        End If
      End Do
      If (exists .EQV. .FALSE.) Then
        mpi_list(counter) = alf(i,3)
        counter = counter + 1
      End If
      exists = .FALSE.
    End If
  End If
End Do
```

Figure 5: fflux_calc_forces code to compute the local ALF as well as the mpi_list

Figure 5 shows that code that turns the global indexed ALF into the locally indexed **alf(:,:)** array. The global indexing never changes whereas the local indexing may change each timestep (note: in reality only some of the local indices will change each step). The local indexing only arises in the case that MPI is switched on. This code also constructs **mpi_list** which stores indices of atoms that need "special attention" for the MPI. More on this can be found in the parallelisation section but, here it suffices to say that this list facilitates the computation and subsequent communication of forces on atoms that belong to molecules that are split across the boundary between 2 (or more) domains.

nlast	DL_POLY variable that is equal to the number of local atom + the number of halo atoms. Halos are required for periodic boundary conditions and/or MPI.
local_index	A DL_POLY function that takes a global index and returns a local index.

```
! Virial calculation corrected for PBC's + MPI
  Do i=1,natms
    !Check if i is the heaviest atom in the ALF, if so no position correction
    !needed
    If ((i .EQ. alf(alf(i,2),2)) .AND. (i .EQ. alf(alf(i,3),2))) Then

      !Standard f.r for i, no correction as i is in main cell by definition
      virmet = virmet - ((Localfxx(i)*xxx(i)) + (Localfyy(i)*yyy(i)) + (Localfzz(i)*zzz(i)))

      !Update stress tensor components.
      strs1 = strs1 + xxx(i)*Localfxx(i)
      strs2 = strs2 + xxx(i)*Localfyy(i)
      strs3 = strs3 + xxx(i)*Localfzz(i)
      strs5 = strs5 + yyy(i)*Localfyy(i)
      strs6 = strs6 + yyy(i)*Localfzz(i)
      strs9 = strs9 + zzz(i)*Localfzz(i)

    Else

      xt = xxx(i) - xxx(alf(i,2))
      yt = yyy(i) - yyy(alf(i,2))
      zt = zzz(i) - zzz(alf(i,2))

      xtmp = xxx(alf(i,2))
      ytmp = yyy(alf(i,2))
      ztmp = zzz(alf(i,2))
      Call images_s(imcon, cell, xtmp,ytmp,ztmp)
      xt = xxx(i) - xtmp
      yt = yyy(i) - ytmp
      zt = zzz(i) - ztmp

      Call images_s(imcon, cell, xt,yt,zt)
      xt = xt + xtmp; yt = yt + ytmp; zt = zt + ztmp;
      !Virial contribution
      virmet = virmet - ((Localfxx(i)*xt) + (Localfyy(i)*yt) + (Localfzz(i)*zt))

      !Update stress tensor components.
      strs1 = strs1 + xt*Localfxx(i)
      strs2 = strs2 + xt*Localfyy(i)
      strs3 = strs3 + xt*Localfzz(i)
      strs5 = strs5 + yt*Localfyy(i)
      strs6 = strs6 + yt*Localfzz(i)
      strs9 = strs9 + zt*Localfzz(i)

    End If
  End Do
```

Figure 6: fflux_calc_forces code to copmute IQA force contribution to stress tensor.

Figure 6 shows how the IQA contribtion to the stress tensor is computed. There are a couple of important things to note. Firstly, the virial is computed using the global virial equation, not the pairwise equation that is typically used. Secondly, in the case of PBC's and/or MPI it is important that the book-keeping is done correctly in the case that a molecule is split across the boundary of the box (for PBC's) or a domain (MPI). For more information on this see.[3, 4] Note that this code is the same as is found in fflux_ewald.f90 which computes the contribution to the stress tensor due to the FFLUX specific electrostatic forces.

3.5 fflux_calc_features

This subroutine is relatively self explanatory, it simply calculates all of the features for each of the atoms. This is necesary in order to then make predictions (done in fflux_predict_value). The functions called in this subroutine can all be found in fflux_module and the maths can

be found in reference [5]. Note that, for PBC's, the distances fed into this subroutine must be wrapped, this is done in fflux_calc_forces.f90. Note that features are computed for local and halo atoms.

features(:,:)	Array that stores the features of each atom. Size is (**mxatms**,**maxFeatures**). Note that currently **features** is allocated once at the start hence the first dimension is of size **mxatms**. This is likely much larger than will ever be needed.
mxatms	A DL_POLY variable that is an estimate (calculated based on initial density of the system + safety margins) of the maximum number of atoms that could ever be in a single cell/domain.
lxx(:,:)	Stores the first row of the C Matrix i.e. C_{11}, C_{12}, C_{13} for each atom. Similalry for **lyy** and **lzz** with the second and third rows respectively.
dCxx(i,j,a,b)	This array stores the derivatives of the first row of the C matrix (i.e. **lxx**) with respect to global atomic positions. The C matrix belongs to atom i and j runs from 1 to 3 denoting the position in atom i's ALF of the atom whose coordinates the derivative is with respect to. The third dimension runs from $a = 1-3$ where values 1, 2 and 3 represent derivatives with respect to the x, y and z components of position respectively. The final dimension runs from $b = 1-3$ which represents the component of the C matrix being differentiated i.e. C_{1b} in the case of **dCxx**. The arrays **dCyy** and **dCzz** are defined in a similar way for the second and third rows of the C matrix.

3.6 fflux_predict_value

This is a relatively simple routine that implements the RBF kernel in as well as its derivative with respect to features in order to make predictions of properties and compute forces respectively. The derivative of the kernel with respect to features is used as part of a chain rule to then compute forces see reference [5]. The predicted value is accumulated in **Q_est** and the derivatives in **dQ_df(:)**. With reference to the maths in reference [5] it is simple to see what is being done in this routine.

3.7 fflux_derivs

This subroutine computes the derivative of a predicted property Q with respect to global, atomic Cartesian coordinates (needed for force calculations). The derivative is done via a chain rule, making use of **dQ_df** that is computed in fflux_predict_value.f90 as well as the routines in fflux_module.f90 that compute the derivatives of features with respect to atomic coordinates. With reference to the maths in reference [5] this routine should be relatively self explanatory.

3.8 fflux_calc_mp_forces

This routine is called if the cluster method of electrostatics is done. This routine is **NOT** to be used with periodic boundary conditions or in parallel i.e. it is for small clusters only. The routine is called for each pair of atoms that interacts electrostatically. In the routine, there is a loop over **nPred_A** and **nPred_B** which are the predicted properties (multipole moment components) for atom A and B i.e. in total there is 4-deep loop over atoms i, j and properties Q^A and Q^B. The energy of the interaction between a pair of atoms is calculated as,

$$E_{ij} = \sum_{A,B} Q_i^A T^{AB} Q_j^B \tag{1}$$

where T^{AB} is the interaction tensor. Equation 1 as well as its derivative is implemented in this routine (see references [6, 7] for the full mathematical details).

3.9 fflux_ewald

This subroutine is the "hub" for everything related to the smooth particle mesh Ewald (SPME) summation in DL_FFLUX. Initially, multipole moments are prepared and then they are fed into the modified Ewald routines for each part of the Ewald sum: reciprocal, real and excluded. Note that the self interaction is typically computed in the reciprocal sum routine. Much of fflux_ewald (as well as fflux_rotate_moments and fflux_tensor_conversion) is concerned with computing and transforming multipole moments as well as the derivatives of multipole moments. Multipole moments are predicted in spherical tensor form in the atomic local frame (ALF). As such the following procedure must be applied:

1. Predict multipole moments and their derivatives with respect to atomic positions in local, spherical tensors - stored in **SphericalPoles(:)** and **dQ_da(:,:,:,:)** respectively.

2. Transform moments and their derivatives into Cartesian tensor form. For the derivatives this transformation is done in fflux_ewald. For the moments, this transformation is done inside of fflux_tensor_conversion (see Figure 1).

3. Transform (i.e. rotate) moments and derivatives into the global frame. For multipole moments this is done in fflux_rotate_moments. For the derivatives this is done in fflux_global_cmoment_derivs (called from within the code as fflux_cmoment_derivs).

4. The final thing to account for is the degeneracy in the unidimensional formalism that is used by DL_POLY (see references [8, 9]). This amounts to multiplying certain multipole moment components (and their derivatives) by certain factors. This is done when the transformation from the local to global frame is done.

dQ_da(j,i,l,a)	Array that stores the local Cartesian derivatives of the multipole moment components. The derivative is of a moment on atom i with respect to the coordinates of the atom that is the position j in the atom i's ALF (for $j > 3$, the index denotes the position in the non_alf). The 3rd dimension, l denotes which multipole moment component is being differentiated (see DL_POLY manual for the ordering) and a runs from 1-3 which represents the x, y and z derivatives respectively i.e. $\partial Q_i^l / \partial a_j$
dQG_da(:,:,:,:)	Array that stores the global Cartesian derivatives of the multipole moment components. Indexed in the same was as **dQ_da**.
mplgfr(l,i)	DL_POLY variable that stores global frame multipole moments, this is filled with the predicted + transformed FFLUX moments (note that for point charge only the array **chge(:)** is used instead). The first dimension runs over the number of multipole moment components (see DL_POLY manual for the ordering) and the second dimension runs over the atoms.
efx(:)	Array that stores the x-component of the extra forces that are computed in flexible SPME (similarly for **efy(:)** and **efz(:)**). Forces are stored separately in order to compute the stress tensor contributions that arise from these extra forces.
fxc(:)	Array that stores the forces that need to be communicated across domains in the case of MPI where a molecule is split across domains (see parallelisation section for more information).

3.10 fflux_rotate_moments

This routine implements Equation 2 in order to rotate an arbitrary rank tensor. We want to rotate the multipole from the ALF which is defined by the C matrix into the global frame. As such the rotation matrix that we need is the inverse of the C matrix (C is unitary and so the inverse is equal to the transpose).

$$T_{\alpha\beta\gamma\ldots} = R_{\alpha'\alpha} R_{\beta'\beta} R_{\gamma'\gamma} \ldots T'_{\alpha'\beta'\gamma'\ldots} \tag{2}$$

which in our specific case becomes,

$$Q^{Global}_{\alpha\beta\gamma\ldots} = C^T_{\alpha'\alpha} C^T_{\beta'\beta} C^T_{\gamma'\gamma} \ldots Q^{ALF}_{\alpha'\beta'\gamma'\ldots} \tag{3}$$

```
Allocate(ixyz0(1:mxatms))
ixyz0(1:nlast) = ixyz(1:nlast)
mlast=natms

Call mpoles_rotmat_export(-1,mlast,ixyz0,CartPoles)
Call mpoles_rotmat_export( 1,mlast,ixyz0,CartPoles)

Call mpoles_rotmat_export(-2,mlast,ixyz0,CartPoles)
Call mpoles_rotmat_export( 2,mlast,ixyz0,CartPoles)

Call mpoles_rotmat_export(-3,mlast,ixyz0,CartPoles)
Call mpoles_rotmat_export( 3,mlast,ixyz0,CartPoles)
```

Figure 7: fflux_rotate_moments code.

The code in Figure 7 is required in the case of PBC's and/or MPI. It fills in the multipole moments for the halo atoms given the local atom moments.

3.11 fflux_tensor_conversion

This subroutine is fairly self explanatory. Multipole moments are converted from spherical to Cartesian tensors. See Appendix E of Stone [7] for the Equations. **IMPORTANT NOTE: 1st Edition has errors, use the 2nd edition only!!!!!!**.

IMPORTANT NOTE: DL_POLY assumes that the moments it receives will have already been multiplied by the relevant factors depending on whether the moments are traceless or traced. The conversions in Stone generate traceless Cartesian moments and so the relevant pre-factors must be applied, this is done as part of the conversion in this routine.

3.12 fflux_global_cmoment_derivs

This routine computes the global derivatives of Cartesian multipole moments with respect to atomic positions. The derivative is essentially the derivative of Equation 2 via a chain rule. This gets progressively more complex as the rank of the multipole moment increases i.e. the number of rotation matrices and thus terms in the chain rule increases. The code for each of the multipole moments i.e. dipole, quadrupole etc. follows a similar pattern. An example of the maths involved for the octupole moment (one of the most complex) is given here,

$$O^G_{\beta\gamma\delta} = \sum_\eta C^T_{\beta\eta} \sum_\varepsilon C^T_{\gamma\varepsilon} \sum_\nu C^T_{\delta\nu} O^L_{\eta\varepsilon\nu} \tag{4}$$

$$O^G_{\beta\gamma\delta} = \sum_\eta C_{\eta\beta} \sum_\varepsilon C_{\varepsilon\gamma} \sum_\nu C_{\nu\delta} O^L_{\eta\varepsilon\nu} \tag{5}$$

The derivative is with respect to an atomic position $\alpha = x, y, z$ is therefore,

$$\begin{aligned}\frac{\partial O^G_{\beta\gamma\delta}}{\partial\alpha} &= \sum_\eta C_{\eta\beta}\sum_\varepsilon C_{\varepsilon\gamma}\sum_\nu C_{\nu\delta}\frac{\partial O^L_{\eta\varepsilon\nu}}{\partial\alpha} + \sum_\eta C_{\eta\beta}\sum_\varepsilon C_{\varepsilon\gamma}\sum_\nu \frac{\partial C_{\nu\delta}}{\partial\alpha}O^L_{\eta\varepsilon\nu} \\ &+ \sum_\eta C_{\eta\beta}\sum_\varepsilon \frac{\partial C_{\varepsilon\gamma}}{\partial\alpha}\sum_\nu C_{\nu\delta}O^L_{\eta\varepsilon\nu} + \sum_\eta \frac{\partial C_{\eta\beta}}{\partial\alpha}\sum_\varepsilon C_{\varepsilon\gamma}\sum_\nu C_{\nu\delta}O^L_{\eta\varepsilon\nu}\end{aligned} \tag{6}$$

Some terms can be collected (this is exploited in the code) as follows,

$$\begin{aligned}\frac{\partial O^G_{\beta\gamma\delta}}{\partial\alpha} &= \sum_\eta C_{\eta\beta}\sum_\varepsilon C_{\varepsilon\gamma}\sum_\nu \left(C_{\nu\delta}\frac{\partial O^L_{\eta\varepsilon\nu}}{\partial\alpha} + \frac{\partial C_{\nu\delta}}{\partial\alpha}O^L_{\eta\varepsilon\nu}\right) \\ &+ \sum_\eta C_{\eta\beta}\sum_\varepsilon \frac{\partial C_{\varepsilon\gamma}}{\partial\alpha}\sum_\nu C_{\nu\delta}O^L_{\eta\varepsilon\nu} + \sum_\eta \frac{\partial C_{\eta\beta}}{\partial\alpha}\sum_\varepsilon C_{\varepsilon\gamma}\sum_\nu C_{\nu\delta}O^L_{\eta\varepsilon\nu}\end{aligned} \tag{7}$$

3.13 fflux_comms

This subroutine as well as fflux_export_data should be read in conjunction with the parallelisation section in order to be properly understood. This subroutine works out which domain(s) each of the force terms need to be sent to.

3.14 fflux_export_data

Routine that facilitates the MPI communication in DL_FFLUX. The buffer is packed to be sent and then sends/receives and unpacks buffers. Note that prior to sending buffers, the size of buffers must be exchanged.

4 Support for multiple kernels

This is a recent addition, DL_FFLUX has been extended to support more than one kernel (previously just the RBF kernel was usable). Currently, only the periodic kernel has been implemented as an extra option but, the architecture of the code should allow for relatively easy addition of more kernels in the future.

The Model type now has a property **kernels(:)** which is an array of integers. The array is of length **nFeatures** i.e. for each feature it stores an integer that corresponds to which kernel acts on that feature (dimension). In fflux_predict_values, for each feature there is a check which kernel should be used (done with an If statement at the moment but probably needs to be made smarter in the future). The list of allowed kernel names is set in fflux_initialisation.f90.

5 Parallelisation

Firstly, a note on indexing in the case of parallelisation with domain decomposition MPI. DL_POLY and therefore DL_FFLUX opts for local indices. There is always a global indexing of atoms that never changes. Each domain is a portion of the overall system and

as such contains some subset of atoms. These atoms will likely have some relatively random assortment of global indices. As such, a local indexing is created that runs from 1 to **natms** where **natms** has been re-defined for each domain so that it is no longer the total number of atoms in the system but instead the total number of atoms local to a given domain. Note that this allows the code to remain unchanged (e.g. loops over natms etc.) when run in serial and in parallel. This is convenient although it does introduce a problem of book-keeping.

The many-body nature of the forces in DL_FFLUX introduces a problem when it comes to parallelisation. Consider the special case (although a special case, this will invariably happen at some point in a bulk simulation) that a molecule is split across two domains.

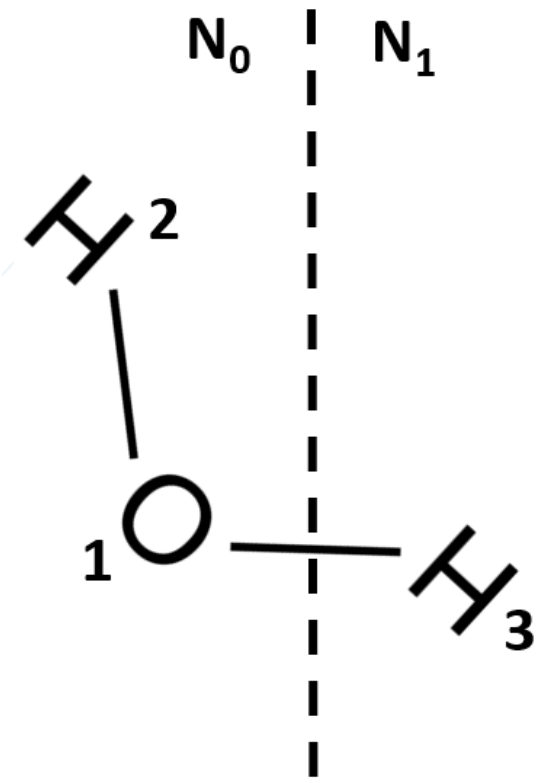

Figure 8: Demonstrates the case of a water molecule split across two domains.

Figure 8 shows a water molecule split across two domains, N_0 and N_1. The dashed line shows the divide between domains. Atoms 1 and 2 are local to N_0 and atom 3 is local to domain N_1. In order to compute the electrostatic forces on atom 1 that arise from the dependence of multipole moments on atomic positions we must compute the derivatives of the multipole moments Q, with respect to the position of atom 1 α_1,

$$-F_{\alpha}^{1} \propto \sum_{l} \left(\frac{\partial Q_1^l}{\partial \alpha_1} + \frac{\partial Q_2^l}{\partial \alpha_1} + \frac{\partial Q_3^l}{\partial \alpha_1} \right) \tag{8}$$

Note that for clarity the full expression for the force is not written in Equation 8 hence the proportional sign rather than an equality. For a given domain, only the derivatives of multipole moments of atoms that are local to that domain are computed (this is done in fflux_ewald). In other words N_0 has access to the first two terms in Equation 8 but not the

third. Domain N_1 has the information necessary to compute the 3rd term. The solution is that domain N_1 will compute this 3rd term and then send it to domain N_0 using an MPI communication.

The implementation of this solution requires multiple modifications to the code. Firstly, we need to figure out which, if any, molecules are split across a domain boundary (note this generalises beyond a molecule split across just 2 domains to a molecule split across an arbitrary number of domains). This is done each timestep in fflux_calc_forces with the code shown in Figure 5. The situation of a split molecule is identified if the atom in question has an index $\leq$ *natms* i.e. it is local to the domain but one of the members of its ALF has an index that is $>$ *natms* i.e. it is not local to the domain.

Once the atoms that require extra force computations to be sent to neighbouring domains have been identified then it is relatively simple to compute these forces (you will noticed loops over **mpi_list** in the Ewald routines). These forces are stored separately and then sent to the correct domain (done in fflux_comms and fflux_export_data).

The following arrays in DL_FFLUX are indexed globally and so all accesses to elements of these arrays should be done using global indices:

model_list()
non_alf_atms()
globalAlf()

6 Important Notes

This sections contains notes about things that are not bugs as such but should probably be fixed/changed at some point.

- Currently the order of the model names in the CONFIG file must be in increasing numerical order.
- Cluster electrostatics should **NOT** be used with PBC's or in parallel. It is only for small clusters run in serial.
- When the dipole printing is done, the writes are not currently MPI safe so the code must be run in serial.
- If, when running in parallel you get the error "link cell algorithm in contention with SPME sum precision" then increase the precision of the SPME sum (e.g. from $1d-6$ to $1d-7$) and it should work.
- If using OpenMP (OMP), never set OMP_PROC_BIND to true.
- Currently longest allowed model name is 64 characters. This should be way more than enough but if not then it can be changed in fflux_module.f90.

References

[1] B. C. B. Symons, M. K. Bane, and P. L. A. Popelier, "DL_FFLUX: A parallel, quantum chemical topology force field," *J. Chem. Theory Comput.*, vol. 17, pp. 7043–7055, 2021.

[2] B. C. B. Symons and P. L. A. Popelier, "Flexible multipole moments in smooth particle mesh ewald," *Unpublished*, 2004.

[3] A. P. Thompson, S. J. Plimpton, and W. Mattson, "General formulation of pressure and stress tensor for arbitrary many-body interaction potentials under periodic boundary conditions," *J. Chem. Phys.*, vol. 131, 2009.

[4] A. Konovalov, B. C. B. Symons, and P. L. A. Popelier, "On the many-body nature of intramolecular forces in fflux and its implications," *J. Comput. Chem.*, vol. 42, pp. 107–116, 2020.

[5] J. C. R. Thacker, A. L. Wilson, Z. E. Hughes, M. J. Burn, P. I. Maxwell, and P. L. A. Popelier, "Towards the simulation of biomolecules: optimisation of peptide-capped glycine using FFLUX," *Molecular Simulation*, vol. 44, pp. 881–890, 2018.

[6] M. J. L. Mills and P. L. A. Popelier, "Electrostatic forces: Formulas for the first derivatives of a polarizable, anisotropic electrostatic potential energy function based on machine learning," *J. Chem. Theory Comput.*, vol. 10, pp. 3840–3856, 2014.

[7] A. J. Stone, *The Theory of Intermolecular Forces*. Oxford University Press, 1996.

[8] H. A. Boateng, I.T. Todorov, "Arbitrary order permanent cartesian multipolar electrostatic interactions," *J. Chem. Phys.*, vol. 142, 2015.

[9] C. Sagui, L. G. Pedersen, and T. A. Darden, "Towards an accurate representation of electrostatics in classical force fields: efficient implementation of multipolar interactions in biomolecular simulations," *J. Chem. Phys.*, vol. 120, pp. 73–87, 2004.