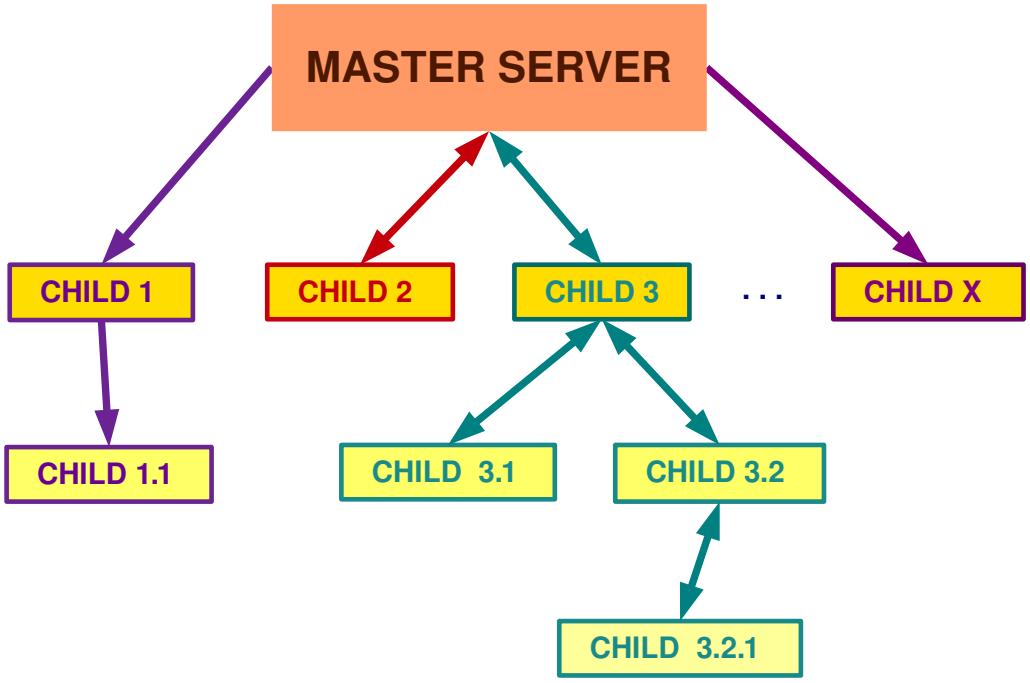


Multi-Model-Driver (MMD) Version 2.0 Library Manual

Astrid Kerkweg[1,3] **& Patrick Jöckel**[2]

[1] Institute for Atmospheric Physics
University of Mainz, 55099 Mainz, Germany
[2] Deutsches Zentrum für Luft-und Raumfahrt (DLR),
Institut für Physik der Atmosphäre,
Oberpfaffenhofen, D-82234 Weßling, Germany
patrick.joeckel@dlr.de
[3] Meteorological Institute
University of Bonn, 53121 Bonn, Germany
kerkweg@uni-bonn.de

This manual is available as electronic supplement of our article "The on-line coupled atmospheric chemistry model system MECO(n) Part 5: Expanding the Multi-Model-Driver (MMD v2.0) for 2-way data exchange including data interpolation via GRID (v1.1) " in Geosci. Model Dev. (2016), available at: http://www.geosci-model-dev.net

Date: January 19, 2017

Contents

1 Introduction

This is the documentation of the second release of the Multi-Model-Driver (MMD) library. The first MMD version was written following a Client-Server approach, where the Server (driving model) provides data to a Client model. The most prominent application is a global model driving a regional model. For the data exchange the sending and the receiving side must be distinguished. Therefore the model instance sending the information was called "server" and the model receiving data was named "client". The current version of the Library is extended to allow data exchange in both directions. Here, the application in mind is the two-way coupling of two models. In order to keep the naming convention of the first version, one model instance (mostly the model determining the overall time control) is still called the "server" or "parent", while the other model is called the "client" or "child". Furthermore the model providing / sending data is called *sending model*[1] (or short the *Sender*) and the model receiving the data the *receiving model* (or short the *Receiver*). The other model instance in a corresponding pair of instances is further denoted as *remote model*, i.e., the *remote model* of the server / parent is the client / child model and vice versa. Note: The terms "model", "model instance" and "instance" do not mean exactly the same thing. A "model" is the model itself (e.g., for the MECO(n) system these are COSMO/MESSy or EMAC). In an MMD coupled system different instances of these models are run concurrently. Thus a "model instance" or "instance" is one realisation of the model configuration within the coupled setup. However, as it is intuitively clear whether a "model" or an "instance" is addressed, we will use these terms synonymously.

MMD consists of two parts:

- the Multi-Model-Driver (MMD) library provides all routines necessary for the data exchange between executables of basemodels, and
- the submodel MMD2WAY organises the data sending to and receiving from the respective remote model and the further processing of the data.

The MMD library manages the communication between the remote models. The submodel MMD2WAY controls the data exchange via namelists. It is split into two sub-submodels MMD2WAY_PARENT and MMD2WAY_CHILD[2]. The sub-submodel MMD2WAY_PARENT comprises the data management required for a parent submodel, while MMD2WAY_CHILD deals with the data management required by a child model.

In this manual we focus on the technical structure and functionality of the MMD library. A description of MMD2WAY is provided within the "MMD user manual", available in the same electronic supplement as this manual. The MMD library manages the data exchange very efficiently, as the field exchange during the time integration is implemented as point-to-point, single-sided, non-blocking MPI[3] communication.

The MMD library is implemented in a way that an arbitrary number of model instances can be run concurrently within the same MPI environment. Each instance can be parent for an arbitrary number (including zero) of other instances and is child of exactly one instance. The only exception is the global or "coarsest" instance of the setup. This one drives all the other instances (directly or indirectly via other child models), but is not child model of any other model itself. This model is called *master parent* or *patriarch*. The figure on the front page of the manual gives an example for a possible model cascade. Here, the *patriarch* "serves" four child models (CHILD 1,2,3 and X). Two of these child models (CHILD 2 and 3) send data back to their parent, which is indicated by the double-ended arrows. CHILD 1 and 3 are parent for other child models as well. CHILD 1 has one child model (CHILD 1.1) and CHILD 3 serves two child models (CHILD 3.1 and CHILD 3.2), which both send back to CHILD 3. CHILD 3.2 is parent model for CHILD 3.2.1 and receives data from it. The child models of a parent model are completely independent of each other.

The only restrictions for such a model cascade are those usually valid for the nesting of limited-area models into coarser models. For instance, each child model domain, including an additional boundary required for interpolations from the coarser to the finer grid, has to be embedded entirely into the parent domain. These

[1] Throughout the manual some words are written in italics. Their meaning is explained in the appendix of the manual.

[2] Remark for those readers familiar with the first version of MMD: MMD2WAY_PARENT and MMD2WAY_CHILD are, with respect to the 1-way coupling, identical to MMDSERV and MMDCLNT, respectively.

[3] Message Passing Interface

requirements become more complex, if the same data are sent back from different child models to the same parent model. There is no interference as long as the child model domains do not overlap. If they do, the data from the childrens are applied in the order of the child instances. However, if different data are exchanged the overlap is technically no problem. Anyhow, it is left to the user to construct a scientifically meaningful setup. The library "only" provides the infrastructure to exchange the data.

The MMD library is mainly written in Fortran95 and partly in C. On the one hanf, C is required, as the data exchange during the time loop is implemented as single-sided communication. The buffer for the data exchange is allocated by the MPI subroutine `MPI_alloc_mem` which works properly only in C, as it hands back the memory address of the buffer, which cannot be used in Fortran95. On the other hand the library must be able to handle Fortran95 POINTERs which – in contrast to C POINTERs – do not have to be consecutive in memory.

The MMD library comprises mainly parent and child model specific routines organised in modules USEd by the MMD2WAY submodel, and it contains routines USEd by the parent and child basemodels directly. The names of the modules and the subroutines or functions indicate, whether the language is C or Fortran95. C files or functions are prefixed with `'mmdc_'`, while Fortran95 files or routines begin with `'mmd_'`. Figure 1 illustrates the dependencies of the MMD library modules and their internal hierarchy. The arrows point into the direction of USagE. The Fortran95 part of the library (bluish colours in Fig. 1) comprises six modules:

- `mmd_child.f90`, `mmd_parent.f90` and `mmd_test.f90` are directly used by the child and parent submodels (MMD2WAY_CHILD and MMD2WAY_PARENT, respectively).

  All three module files in turn USE the three modules `mmd_mpi_wrapper.f90`, `mmd_handle_communicator.f90` and `mmd_utilities.f90`.

- `mmd_mpi_wrapper.f90` supplies high level interface routines, which simplify the communication between parent and child model and offer the possibility for alternative communication implementations without the need for changing the interfaces.

- `mmd_handle_communicator.f90` contains the subroutines for the MPI-communicator definitions in the coupled system.

- `mmd_utilities.f90` hosts the definition of the Fortran95 structures storing the information about the MPI-communicators and the Fortran95 structure keeping the information and data of the *exchange fields*.

The C part of the library (reddish colours in Fig. 1) contains five files:

- `mmdc_child.c` and `mmdc_parent.c` contain the child model and parent model specific functions for the data exchange management.

- `mmdc_util.c` supplies the declaration of the data structures.

- All three C modules access the header file `mmdc_utils.h`, which itself uses the header file `cfortran.h` for C to Fortran95 binding.

For the sake of a better readability only the file names without the language suffixes (`.f90` or `.c`) are used in the remaining text.

The next section provides an overview of the MMD library by describing the typical call sequence of the library routines. More detailed explanations are provided in Sect. 3. Subsect. 3.1 describes the Fortran95 routines, Subsect. 3.2 the C functions of the MMD library. Last but not least, Subsect. 3.3 provides an example to illustrate the data exchange.

2 The call sequence of the MMD library

This section illustrates the interaction and the call sequence of the MMD library subroutines and functions, which are described in detail in Sect. 3. Fig. 2 shows the call tree. The yellow area indicates the initialisation phase of a simulation with MMD coupled models. The cyan part highlights those routines called during the

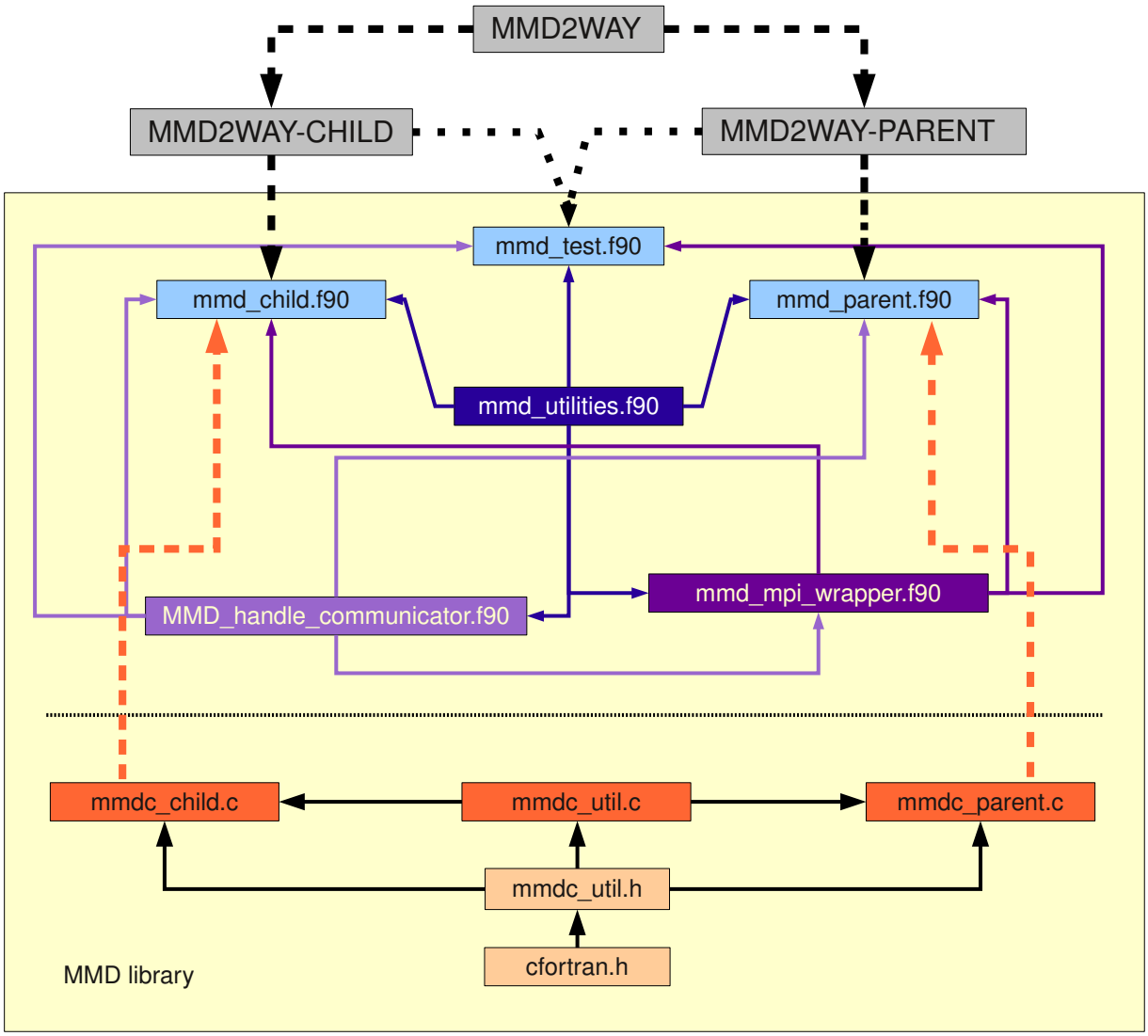


Figure 1: Hierarchy of the MMD module files. Bluish colours indicate modules written in Fortran95, reddish colours denote C files. Arrows point in the direction of USagE, e.g., `mmd_utilities.f90` is used by all other Fortran95 modules.

integration phase, and the lilac part denotes to the finalisation phase. The left hand side lists the routines called by the child model, the right hand side those called by the parent model. Subroutines on the same level are called concurrently from the corresponding entry points in the child and parent model. Note that the names of the child model specific routines always start with "`MMD_C_`", whereas the names of the parent specific routines begin with "`MMD_P_`". The names of the routines for checking the consistency of the setup of the coupled models start with "`MMD_testC_`" or "`MMD_testP_`" depending on the calling model, child or parent, respectively.

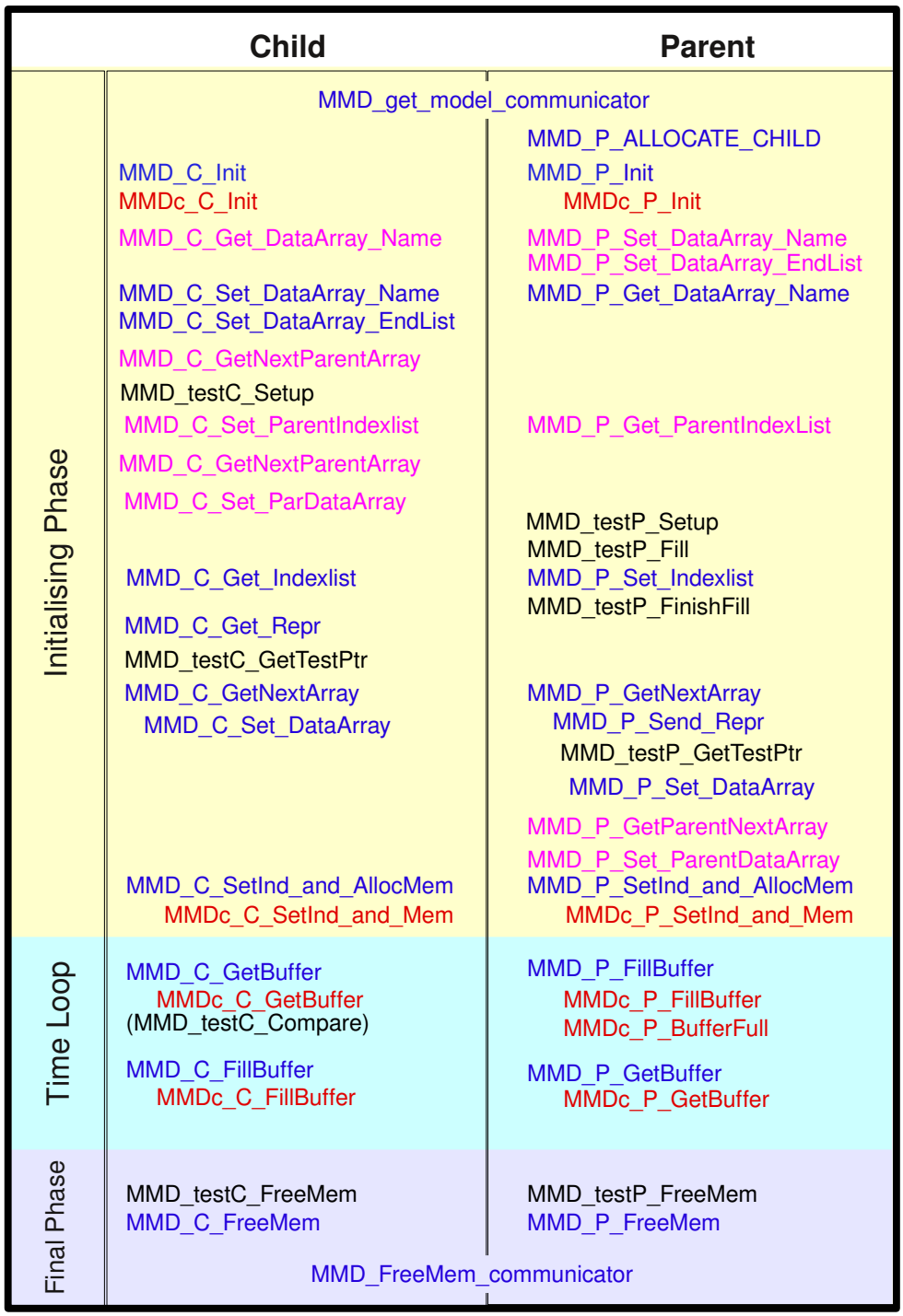


Figure 2: Systematisation of the MMD library: blue are the names of the child and parent model specific Fortran95 routines of the MMD library, red are the C functions, pink are the subroutines added for the 2-way coupling, whereas the black colour indicates the routines required for the test setup (written in Fortran95).

2.1 The initialisation phase

At the very beginning, the MPI-communicators of the different models/parallel processes are determined. In addition to the "global" MPI-communicator `MPI_comm_world` containing all process entities[4] (PEs), a group communicator is set up for each model. This is achieved by the subroutine `MMD_get_model_communicator`. It is called during the MPI setup procedure of all models. Within this subroutine the namelist file `MMD_layout.nml`, which contains all information defining the coupling layout of all instances for the current simulation, is read by the process (PE) with `m_world_rank = 0` and broadcasted to all other processes. Afterwards, the communicators are defined based on the namelist information, i.e., a group of PEs is associated to each model instance and the corresponding group communicator is defined. The subroutines `MMD_get_model_communicator` and `MMD_FreeMem_Communicator` are the only subroutines, which are called directly from the basemodels. All other routines are called from the MESSy submodel MMD2WAY carrying out the coupling.

- In the parent model specific subroutine `MMD_P_Allocate_Child` the structure `Child`, which components contain all required information about the child models and about the data requested by the respective child model, is allocated to the actual number of child models of each individual parent models. For instance, in the example on the front page the *patriarch* requires space for 4 child models, whereas CHILD 1 provides data to only one child model. In case of 2-way coupling, additionally the structure `Parent` is allocated also to the number of child models. The components of this structure hold all information required for sending back data from the child models. Additionally, arrays for the buffer length of the exchange data (`ChldBL` and `PArBL`) are allocated to the number of child models.

- In the subroutines `MMD_P_Init` and `MMD_C_Init` the MMD internal structure components containing the information about the *remote model(s)* are allocated to the correct size according to the coupling layout. Subsequently, the structure components are preset with default values.

  Additionally, the setup routines for the C-core of MMD, `MMDc_C_Init` and `MMDc_P_Init`, are called. Within these routines the communicators for the communication on the C language level are constructed, and the information about the size (number of processes occupied by the *remote model*) is handed back to the Fortran95 part of the library.

- As a first step on the way to the data exchange between the models, the parent and the child models provide lists of parent and child model *channel* and *channel object* names of the required data arrays and the corresponding *respresentation* to the MMD library by calling the subroutines `MMD_P_Set_ParDataArray_Name` and `MMD_C_Set_DataArray_Name`, respectively.

  Within the library routine `MMD_P_Set_ParDataArray_Name` the *channel* and *channel object* names of the fields required by the parent model are stored within the structure variable `Parent`, which is of TYPE `ExchDataDef`(see Sect. 3.1.1). `Parent` contains the overall coupling information for the receiving parent model part of the library. Additionally, this subroutines broadcasts the respective child model *channel* and *channel object* names and their *representation*s to the child model. On the child model side the data is received and processed within the subroutine `MMD_C_Get_ParDataArray_Name`. Here, the *channel* and *channel object* names of the field sent from the child to the parent model and their *representation*s are stored in the structure variable `Parent`, which is also of TYPE `ExchDataDef`. These information will be accessed later on using the functions `MMD_P_GetNextParArray` and `MMD_C_GetNextParArray` on parent and child model side, respectively.

  Subsequently, within the library routine `MMD_C_Set_DataArray_Name`, the *channel* and *channel object* names of the fields required by the child model are stored within the structure `Me`, which is of TYPE `ExchDataDef` and contains the overall coupling information required by the receiving child model part of the library. Additionally, within this subroutine the parent model *channel* and *channel object* names and their *representation*s are sent to the parent model. On the parent side the data is received and processed within the subroutine `MMD_P_Get_DataArray_Name`. Here, the *channel* and *channel object* names of the parent model *exchange fields* and their *representation*s are stored in the information structure (`Child`) for the respective child model. These information will be accessed later on, with the use of the functions `MMD_P_GetNextArray` and `MMD_C_GetNextArray` on parent and child model side, respectively.

[4] Here equivalent to MPI tasks.

The subroutines `MMD_P_Set_ParDataArray_Name` and `MMD_C_Set_DataArray_Name` are called for each *channel* and *channel object* name pair independently. The calls to the subroutines `MMD_P_Set_ParDataArray_EndList` and `MMD_C_Set_DataArray_EndList`,respectively, indicate that the list is complete.

- Additional preparations are performed before the data exchange can take place:
  - First, the information is required, which local model PE exchanges which parts of its horizontal grid (including the indices in the respective local grid) with which remote model PE for both data exchange directions independently. Hereafter, this information is refered to as "`index_list`". It is provided by the MESSy submodel MMD2WAY to the MMD library as parameter to the subroutines `MMD_P_Set_Indexlist` and `MMD_C_Set_ParIndexlist` for 1-way and backward coupling, respectively. Within the subroutine `MMD_P_Set_Indexlist` the information for the data exchange from the parent to the child model is processed (see Sect. 3.1.5). Similarly, in `MMD_C_Set_ParIndexlist` processes the information for the exchange between child and parent model. In these subroutines, the indexlists are evaluated and sent to the respective remote model. The remote model receives and processes this list within the subroutines `MMD_C_Get_Indexlist` and `MMD_P_Get_ParIndexlist`, respectively. For further details see the description of the subroutines `MMD_P_Set_Indexlist` in Sect. 3.1.5 and `MMD_C_Set_ParIndexlist` in Sect. 3.1.4.
  - Second, a test is initiated for checking the data exchange from the parent to the child model. All subroutines with names starting with `'MMD_test'` contribute to this consistency check.
  - Third, the connection between the *channel* and *channel object* names to the respective Fortran95-POINTERs need to be established:
    - Data exchange from parent to child model:
      - On the parent model side, the list of the names of the *exchange fields* established by the subroutine `MMD_P_Get_DataArray_Name` is provided to MMD2WAY_PARENT by the subroutine `MMD_P_GetNextArray`. For each of the required fields the MESSy submodel MMD2WAY_PARENT provides a 4D-POINTER to the memory of the respective *exchange field* as parameter to the subroutine `MMD_P_Set_DataArray`. Additionally, the local *dimensions* and the *axis string* related to this particular field are parameter to this subroutine. This information is stored in the respective structure components and analysed: the *dimensions* determine the memory size (buffer length) required for the data exchange. The *axis string* is used later during the integration to re-arrange the data (before the actual data exchange) into a 1D-array. The sum over all individual buffer sizes is the buffer length required for the data exchange of all data fields from each individual parent PE to all child model PEs. This number is used to actually allocate the required memory with the C function `MMDc_P_SetInd_and_Mem`, which is called via the Fortran95 subroutine `MMD_P_SetInd_and_AllocMem`.
      - For the child model the provision of the data POINTER via the subroutines and functions `MMD_C_Get_DataArray_Name`, `MMD_C_GetNextArray` and `MMD_C_Set_DataArray` works similarly. The *dimension* and *axis string* information are used during the integration phase to re-order the 1D-array received from the parent into the 4D data arrays as requested by the child model.
    - Data exchange from child to parent model:
      - On the child model side, the list of the names of the *exchange fields* established by the subroutine `MMD_C_Get_ParDataArray_Name` is provided to MMD2WAY_CHILD by the subroutine `MMD_C_GetNextParArray`. For each of the required fields the MESSy submodel MMD2WAY_CHILD provides a 4D-POINTER to the memory of the respective *exchange field* as parameter to the subroutine `MMD_C_Set_ParDataArray`. Additionally, the local *dimensions* and the *axis string* related to this particular field are parameter to this subroutine. This information is stored in the respective structure components and analysed: the *dimensions* determine the memory size (buffer length) required for the data exchange. The *axis string* is used later during the integration to re-arrange the data (before the actual data exchange) into a 1D-array. The sum over all individual buffer sizes is the buffer length required for the data

exchange of all data fields from each individual Child PE to all Parent PEs. This number is used to actually allocate the required memory with the C function `MMDc_C_SetInd_and_Mem`, which is called via the Fortran95 subroutine `MMD_C_SetInd_and_AllocMem`.

- For the parent model the provision of the data POINTER via the subroutines and functions `MMD_P_Get_DataParArray_Name`, `MMD_P_GetNextParArray` and `MMD_P_Set_ParDataArray` works similarly. The *dimension* and *axis string* information is used during the integration phase to re-order the 1D-array received from the child model into the 4D data arrays as requested by the parent model.

At this point all preparations required for the data exchange are finished and the actual data exchange starts.

2.2 The time loop

As the data exchange was fully prepared in the initialisation phase, the buffers only need to be filled and read within the time loop. Here, the order of MMD library routine calls is important to avoid an MPI deadlock:

1. The parent model has to fill the buffer. This is done by the MMD library subroutines `MMD_P_FillBuffer` and `MMDc_P_FillBuffer` (see Sect. 3.1.5). After the buffer is filled, it is made accessible for the child model by setting a barrier indicating that the buffer is filled.
2. Afterwards this buffer can be read by the child model. Within the subroutines `MMD_C_GetBuffer` and `MMDc_C_GetBuffer` the (1D-)buffers are read and re-arranged to the 4D-data fields according to the corresponding *axis string* and the local *dimensions*. If only 1-way coupling is required, again a barrier is set to indicate to the parent that the buffer was read and can be filled again, i.e., the cycle starts again with 1. (3.+4. are omitted for 1-way coupling).
3. After the buffer is read by the child model, it can be filled with the child model data coupled back to the parent model. This is performed by the subroutines `MMD_C_FillBuffer` and `MMDc_C_FillBuffer`. At the end of this subroutine a barrier is set to indicate that the buffer is filled by the child model and ready for being read by the parent model.
4. In the last step of the cycle in the subroutines `MMD_P_GetBuffer` and `MMDc_P_GetBuffer` the data provided by the child model are read from the parent and re-arranged into the requested 4D-data format according to the corresponding *axis string* and the local *dimensions*. After the reading is finished, the cycle starts again.

To detect errors in the data exchange the subroutine `MMD_testC_Compare` is called after the first data exchange. The transferred test arrays containing the geographical longitudes and latitudes are compared to the original arrays of the child model. If they do not match, the simulation is terminated with an error message.

2.3 The finalisation phase

At the end of the simulation the memory allocated during the initialisation must be deallocated. The test arrays are deallocated within the subroutines `MMD_testC_FreeMem` and `MMD_testS_FreeMem`, respectively. The memory allocated for the MMD library data structures is released in `MMD_C_FreeMem` and `MMD_P_FreeMem` for child and parent model, respectively. Last but not least, the memory allocated by the subroutine `MMD_get_model_communicator` is released within the subroutine `MMD_FreeMem_Communicator`.

3 Detailed library description

In this section the definitions and routines of the MMD library are described in detail. The Fortran95 and the C part are explained in individual subsections. Each of the subsections is split into subsubsections dedicated to one (module) file each. Each subsubsection contains the interface declarations of the routines of the respective module. Their content and usage are described subsequently. The section is closed with an example illustrating the definitions of the index and length variables used in the C and the Fortran95 part.

3.1 The Fortran95 part of the MMD library

The Fortran95 part of the library is the interface between the coupled models (more precisely between the coupling subsubmodels MMD2WAY_CHILD and MMD2WAY_PARENT) and the C routines providing the infrastructure for the data exchange during the integration. The library modules are described in the order of their dependencies. As the three main interface modules `mmd_child.f90`, `mmd_parent.f90` and `mmd_test.f90` are completely independent of each other, they are described in an arbitrary order.

3.1.1 mmd_utilities

`mmd_utilities` is used by all other Fortran95 modules. It provides the definition of the data structures containing all information about the model system setup, the communicators, about the structure of the data, the `index_list` for the data exchange between two models and the definition of some PARAMETERs controlling the coupling procedure:

```
! ******************************************************************
! from messy_main_constants_mem.f90
INTEGER, PARAMETER, PUBLIC  :: MMD_DP = SELECTED_REAL_KIND(12,307)
INTEGER, PARAMETER, PUBLIC  :: MMD_I8 = SELECTED_INT_KIND(14)

INTEGER, PARAMETER          :: DP=MMD_DP

! Length of Data Array Name
INTEGER,PARAMETER, PUBLIC   :: STRLEN_CHANNEL = 23
! Length of Data Array Name
INTEGER,PARAMETER           :: STRLEN_OBJECT  = 55
INTEGER, PARAMETER, PUBLIC  :: STRLEN_MEDIUM  = 24
INTEGER,PARAMETER, PUBLIC   :: STRLEN_ULONG   = 256
! ******************************************************************

! ******************************************************************
! Definition PARAMETER
INTEGER,PARAMETER,PUBLIC :: MMD_ParentIsECHAM = 1
INTEGER,PARAMETER,PUBLIC :: MMD_ParentIsCOSMO = 2
! return status
INTEGER,PARAMETER,PUBLIC :: MMD_STATUS_OK    = 0
INTEGER,PARAMETER,PUBLIC :: MMD_DA_NAME_ERR  = 10

INTEGER,PARAMETER,PUBLIC :: MMD_MAX_MODEL    = 64
! ******************************************************************
```

The first block defines the KIND PARAMETERs and the string lengths required within the library identically to the MESSy definitions. The second block consists of MMD internal settings. In the specific application of the MMD library, the MESSy subsubmodel for the coupling on the child model side (MMD2WAY_CHILD) requires the information about the parent model type (spectral (gaussian grid) = ECHAM = gaussian grid or COSMO = lat-lon grid), as some parts of the interpolation/coupling procedure depend on the parent model type. The INTEGER PARAMETERs `MMD_ParentIsECHAM` and `MMD_ParentIsCOSMO` are used to indicate, whether the parent is ECHAM or COSMO. This list can be expanded by newly introduced models. `MMD_STATUS_OK` and `MMD_DA_NAME_ERR` define some specific error values. `MMD_MAX_MODEL` determines the number of model instances maximally handled by MMD. If more than 64 model instances shall be run in parallel by MMD, this number needs to be increased within the code.

The TYPE `ExchDataDef` includes all information required to associate the correct data, dimensions of data fields and grid points to each other:

```
TYPE ExchDataDef
   ! REMOTE MODEL ID
   INTEGER                              :: RMId       = 0
   ! NUMBER OF REMOTE MODEL PEs
   INTEGER                              :: inter_npes = 0
   ! STRUCTURE FOR EACH PE
   TYPE(PeDef), DIMENSION(:), POINTER :: PEs
   ! INDEX LIST OF PARENT MODEL POINTS
   INTEGER,DIMENSION(:,:),ALLOCATABLE :: index_list_2d
   ! Number of Points in index_list
   INTEGER                              :: NrPoints
   ! ARRAY INFORMATION STRUCTURE (SAME ON ALL PEs)
   TYPE(ArrayDef_list), POINTER       :: Ar         => NULL()
   TYPE(ArrayDef_list), POINTER       :: ArrayStart => NULL()
 END TYPE ExchDataDef
```

This structure defines the setup of exactly one *remote model*. Therefore the child model needs one scalar variable of the type of these structures, whereas each parent needs an array dimensioned by the number of child models.

`RMId` is the identification number (ID) of the *remote model* within the overall MMD setup. It is equal to the instance number, e.g. for the example given in Sect. 3.1.2 and Fig. 5, COSMO/MESSy 3.2 has `RMId=8`. The `RMId` is only used by a parent model.

`inter_npes` gives the number of processing entities (PEs) used by the *remote model*, i.e., for the child model the number of parent PEs is stored, whereas for a parent model the number of child model PEs is required.

During the initialisation of MMD the structure component `PEs` will be dimensioned with `inter_npes` as the structure components of TYPE `PeDef` get specific values for each PE of the *remote model*:

```
TYPE PeDef
   INTEGER                                 :: NrEle   ! Number of Elements
   TYPE(xy_ind), POINTER, DIMENSION(:) :: locInd
END TYPE PeDef
```

`NrEle` is the number of elements exchanged with each individual *remote model* PE[5]: For a *receiving model* `NrEle` is the number of grid points the *receiving model* PE gets from a specific *sending model* PE. For a *sending model* `NrEle` is the number of grid points, which this *sending model* PE provides to a specific *receiving model* PE. `locInd` is dimensioned by `NrEle` and contains the index pairs for each exchanged grid point in the particular local decomposed grid:

```
! Pair of indices in horizontal plane
TYPE xy_ind
   INTEGER :: i
   INTEGER :: j
END TYPE xy_ind
```

Thus, for the *receiving model*, `locInd` contains the index pairs of the local decomposed field as received from the *sending model* (here after denoted *in-field*), whereas for the *sending model* `locInd` stores the indices in its own local decomposed model domain.

Figure 3 together with Table 1 illustrates these dependencies:

Part A of Fig. 3 shows two model domains. The left hand side illustrates the parallel domain decomposition of the *sending model* domain (blue). The *Sender* is run on 4 PEs. The decomposed model domains consist of 5x2 model grid boxes each. The model domain of the *receiving model* (red) was chosen to illustrate the grid association. The *receiving model* is run on 3 PEs. The decomposed domains of PE 0 and PE 1 consist of 2x1

[5]Note: as the library is run in the parallel decomposition of the respective model, `NrEle` and `locInd` are different and specific for each PE.

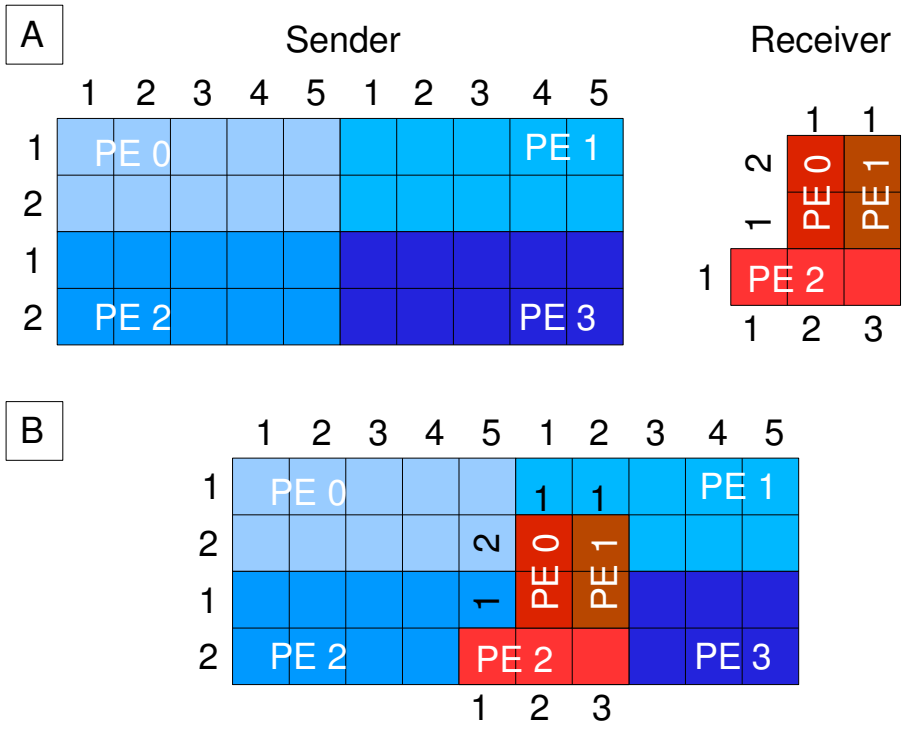


Figure 3: Illustration of possible model domain overlaps and the definition of the variable `NrEle`. A detailed explanation is provided in the text.

Sender PE	*Receiver* PE	NrEle	(x,y) index pairs *Sender*	(x,y,) index pair *Receiver*
0	0	0	-	-
0	1	0	-	-
0	2	0	-	-
1	0	1	(1,2)	(2,1)
1	1	1	(2,2)	(2,1)
1	2	0	-	-
2	0	0	-	-
2	1	0	-	-
2	2	1	(5,2)	(1,1)
3	0	1	(1,1)	(1,1)
3	1	1	(2,1)	(1,1)
3	2	2	(1,2)(2,2)	(2,1)(3,1)

Table 1: Illustration of the data exchange between *sending* and *receiving model* PEs. `NrEle` is the number of elements exchanged, whereas the index pairs show the identification of the grid points in the *sending model* domain and the *in-field* grid points in the *receiving model* domain.

model grid boxes, that of PE 2 of 3x1 grid boxes. Part B of the figure shows the overlap of the *in-field* model domain of the *receiving model* with the *sending model* domain. Table 1 gives the number (`NrEle`) and the index

pairs of exchanged elements for this example[6]: The *Sender* PE 3 sends the data of four grid boxes in total. One element is sent to *Receiver* PE 0 ((1,1)$_{s,PE3}$ → (1,1)$_{r,PE0}$), one element (2,1)$_{s,PE3}$ is sent to *Receiver* PE 1 and *Receiver* PE 2 gets two elements ((1,2)$_{s,PE3}$ and (2,2)$_{s,PE3}$). The *Receiver* PE 2 operates on three grid boxes and needs to get this number of elements. The *Receiver* grid box (1,1)$_{r,PE2}$ on PE 2 is filled by *Sender* PE 2, which sends its grid box (5,1)$_{s,PE2}$. The other two elements are provided by *Sender* PE 3: *Receiver* grid box (2,1)$_{r,PE2}$ gets the data from *Sender* grid box (1,2)$_{s,PE3}$ and *Receiver* grid box (3,1)$_{r,PE2}$ gets the data of *Sender* grid box (2,2)$_{s,PE3}$.

The information, which *Receiver in-field* grid box corresponds to which *sending model* domain grid box is saved for one Child-Parent pair in the structure component `index_list_2d`. `index_list_2d` is only used during the initialisation phase and deallocated afterwards. Further details about `index_list_2d` are provided in Sect. 3.1.5 and Sect. 3.1.4.

The structure component `NrPoints` is only relevant on the *Sender* side of the library. It is the overall number of horizontal elements the current *Sender* PE has to send to all PEs of one *receiving model*. In the example this would be 0 for *Sender* PE 0, 2 for *Sender* PE 1, 1 for *Sender* PE 2 and 4 for *Sender* PE 3.

Last but not least, the structure components `Ar` and `ArrayStart` define a concatenated list, where `ArrayStart` points to the first element of the list and `Ar` to the actual one. The TYPE `ArrayDef_list`

```
TYPE ArrayDef_list
   TYPE(ArrayDef)               :: arrdef
   TYPE(ArrayDef_list), POINTER :: next
END TYPE ArrayDef_list
```

constructs a concatenated list of structure variables of TYPE `ArrayDef`:

```
TYPE ArrayDef
   CHARACTER(LEN=STRLEN_CHANNEL)          :: channel = '' ! Name of Channel
   CHARACTER(LEN=STRLEN_OBJECT)           :: object  = '' ! Name of Object
   CHARACTER(LEN=STRLEN_MEDIUM)           :: repr    = '' ! Representation
                                                          !  of Object
   ! interpolation Method
   INTEGER                                :: interpM
   LOGICAL                                :: l_sentunit
   ! DATA POINTER
   REAL(DP), POINTER, DIMENSION(:,:,:,:)  :: p4 => NULL()

   CHARACTER(LEN=4)      :: dim_order = '    ' ! Order of dimensions
   INTEGER, DIMENSION(4) :: xyzn_dim  = 0      ! index of x,y,z,n dimension
   INTEGER, DIMENSION(4) :: dim       = 0      ! Size of Dimensions
   ! ArrLen and ArrIdx are different on each remote PE
   ! Dimension of Array moved
   INTEGER, DIMENSION(:), POINTER :: ArrLen => NULL()
   ! Start Index of Array moved
   INTEGER, DIMENSION(:), POINTER :: ArrIdx => NULL()
END TYPE ArrayDef
```

This TYPE contains the description of one *exchange field*. For the unambiguous identification of each *exchange field*, `ArrayDef` contains the *channel* name, the *channel object* name and the *representation* name of the object. Additional information, on the interpolation method (`interpM`) and, if the unit of the exchanged field needs to be transferred (`l_sentunit`), are structure components.

`p4` is the POINTER to the respective memory, i.e., to the *in-field* on the *Receiver* side and to the variable on the *Sender* side.

[6]Hereafter, individual horizontal elements are denoted using the syntax (i,j)$_{s\ or\ c,\ PE\ n}$. i and j are the index pair in the respective local parallel decomposed grid, s or r indicates *sending* or *receiving model*, respectively, and the respective PE of the *sending* or *receiving model* model is denoted by *PE n*, with *n* being the number of the PE.

description	representation	dim_order	xyzn_dim	dimension length
COSMO 3d	'GP_3D_MID'	'XYZ-'	(1,2,3,-1)	(ie,je,ke,-1)
ECHAM5 3d	'GP_3D_MID'	'XZY-'	(1,3,2,-1)	(nproma, ngpblks,nlev,-1)
COSMO tracer	-	'XYNZ'	(1,2,4,3)	(ie,je,ntrac,ke)
ECHAM5 tracer	-	'XZNY'	(1,4,2,3)	(nproma,nlev,ntrac, ngpblks)

Table 2: Examples for the definition of the ArrayDef structure components `dim_order`, `xyzn_dim`, `dim`. The last column indicates the variable names of the respective dimension lengths in the respective model, i.e., `ie` and `nproma` are the 'X'-dimension lengths, `je` and `ngpblks` the 'Y'-dimension lengths, `ke` and `nlev` the 'Z' dimension lengths in the COSMO model and the ECHAM5 model, respectively. "-1" denotes an unused rank.

The second block in the structure definition of `ArrayDef` contains the information about the dimensions and the order of the array:

- The CHARACTER string `dim_order` indicates the order of the *dimensions* as string. The first and second horizontal axes are labelled with `'X'` and `'Y'`, respectively. The vertical axis is labelled with `'Z'`. Each additional axis, e.g., number of tracer, number of aerosol modes, etc., is labelled by `'N'`. If an axis is not used the label is `'-'`. `dim_order` is a copy of the *axis string* defined in the CHANNEL submodel[7]. Table 2 illustrates the definition of `dim_order`.
- The INTEGER array `xyzn_dim` provides the information at which rank which dimension can be found. The first entry indicates the rank of the `'X'` dimension, the second that of the `'Y'` dimension, the third the `'Z'` and the fourth the `'N'` dimension. Unused dimensions are labelled with `-1`. (See Table 2 for examples.) The information contained in the two arrays `xyzn_dim` and `dim_order` are redundant. But the INTEGERs are used as indices to directly access the required rank of the data arrays, whereas the string array is easier to handle, when the order of all dimensions needs to be tested. Therefore, both arrays are stored in the `ArrayDef` structure.
- Finally, the INTEGER array `dim` provides the lengths of the respective dimensions. The first element of `dim` gives the length of the first dimension of an array, the second entry the length of the second dimension etc. An example is shown in the last column of Table 2. The structure components of `ArrayDef` discussed so far describe the properties on the current PE and thus are independent of the *remote model* PE.
- The last two structure components (`ArrLen` and `ArrIdx`) are dimensioned with the number of exchanged elements `NrEle`. Hence, they depend on the *remote PE*.
  - `ArrLen` is the length of the respective array, i.e. the product of all array dimensions, where the product of the horizontal dimensions is given by `NrEle`.
  - `ArrIdx` gives the index within the buffer exchanged with each *remote model* PE, where the respective array starts.

  The buffer exchanged between one *sending* and one *receiving model* PE is simply a 1-dimensional array containing all *exchange fields* aligned one after the other. Therefore, `ArrIdx` is used to find the starting point of the respective field within this 1-dimensional array. `ArrLen` contains the information how many elements starting by `ArrIdx` belong to the respective field described by `Ar`.

In addition to the definitions, `mmd_utilities` comprises also one utility routine and one function.

`SUBROUTINE sort_2d_i`		`(array,sort_ind)`	
name	type	intent	description
mandatory arguments:			
array	INTEGER, DIMENSION(:,:)	INOUT	INTEGER array to sort
sort_ind	INTEGER	IN	first rank index of array. The sorting takes place along the second rank only.

[7]The CHANNEL submodel is described in detail in Jöckel et al., GMD, 2010

The subroutine `sort_2d_i` gets a 2-dimensional array and an index as input. Using this index as index for the first rank, it sorts the 2-dimensional array along its second rank from small to large values. This subroutine is used to reorder the `index_list` according to the *Sender* PE numbers and afterwards, for each *Sender* PE according to the *Receiver* PE numbers (see Subsects. 3.1.5 and 3.1.4).

```
Real(kind=DP) FUNCTION get_Wtime              ()
```

The function `get_wtime` reads the system clock. It can be used to measure the time of the data exchange processes.

3.1.2 `mmd_handle_communicator.f90`

The Fortran95 file `mmd_handle_communicator.f90` defines all the variables required for the parallel setup and the communication among models. These are

- status / error flags indicating, if the operation with the MMD library worked

```
! RETURN status
PUBLIC ::                              MMD_STATUS_OK
INTEGER,PARAMETER,PUBLIC          :: MMD_ERROR_NPES = 1
INTEGER,PARAMETER,PUBLIC          :: MMD_ERROR_MPI  = 2
INTEGER,PARAMETER,PUBLIC          :: MMD_STATUS_OFF = 3
```

- The information about the coupling layout, which is read from the namelist file `MMD_layout.nml`.

```
! Define Types
TYPE MMD_layout
  CHARACTER(LEN=5)        :: name
  INTEGER                 :: Parent_Id
  INTEGER                 :: npe
END TYPE MMD_layout

! Coupler Setup
! -------------------------------------
! Coupler Id of this model
INTEGER                                  :: m_my_CPL_Id
! Number of Coupler in layout file
INTEGER                                  :: m_NrOfCpl
!Information of all coupler
TYPE(MMD_layout),dimension(MMD_MAX_MODEL)  :: m_couplers
! -------------------------------------
! Indicates this PE is Parent for Child model with Id ...
INTEGER,DIMENSION(:),POINTER,PUBLIC :: MMD_Parent_for_Child
```

In the overall coupling layout, each model gets a unique Id, used to identify the respective models unambiguously. On each task, the number of the model this task belongs to, is stored in `m_my_CPL_Id`. The overall number of coupled models is saved in `m_NrOfCpl`. The structure variable `m_couplers` contains the information read from the namelist file `MMD_layout.nml`, i.e. the list of coupled models, containing the `name` of each model (at the time being 'echam' or 'cosmo'), the Id of its parent model (`Parent_Id`), and the number of tasks attributed to this model (`npe`).
Last but not least, a list of model Ids (`MMD_Parent_for_Child`) is provided, listing the Ids of those models the parallel task is Parent / Server for.

- the MPI communicators, enabling intra and inter model communication:

```
! Communicator of this model
INTEGER,PUBLIC                              :: m_model_comm
! Communicator to the parent
INTEGER,PUBLIC                              :: m_to_parent_comm
! Communicator to the child(s)
INTEGER,dimension(MMD_MAX_MODEL), PUBLIC    :: m_to_child_comm
```

  `m_model_comm` is the communicator defined for the model the respective task is part of. `m_to_parent_comm` is the communicator required for the current task to communicate with the respective parent model. And `m_to_child_comm` is an array of all communicators required for the communication with each respective child model.

- Additionally, `m_world_npes` provides the number of tasks available in the full parallel setup (`MPI_comm_world` communicator), and `m_world_rank` defines the rank of the current task in this communicator. The same information is required for the model the current task belongs to: `m_model_npes` gives the number of PEs attributed to the respective model, and `m_model_rank` the rank of this PE in `m_model_comm`.

```
INTEGER                             :: m_world_rank
INTEGER                             :: m_world_npes
INTEGER,PUBLIC                      :: m_model_rank
INTEGER,PUBLIC                      :: m_model_npes
```

- Last but not least, `m_ParentType` provides the information about the parent model type:

```
! Parent Type (1 ECHAM, 2 COSMO)
INTEGER,PUBLIC                      :: m_ParentType
```

Additionally, `mmd_handle_communicator.f90` provides those routines required to assign the above listed variables on each process:

- `MMD_get_model_communicator` is a twofold overloaded subroutine. The subroutines `MMD_cag_model_communicator` and `MMD_get_model_communicator` are called by this name.

SUBROUTINE MMD_get_model_communicator (comm [, MMD_status])			
name	type	intent	description
mandatory arguments:			
comm	INTEGER	OUT	MPI-communicator of the calling model
optional arguments:			
MMD_status	INTEGER	OUT	status flag: the presence of the status flag determines which of the two overloaded routines is addressed. If MMD_status is present `MMD_cag_model_communicator` is used.

  - `MMD_get_model_communicator`:
    This subroutine provides the model specific communicator `m_model_comm` to the basemodel calling this subroutine.
  - `MMD_cag_model_communicator`:
    This subroutine performs the MPI setup on which the entire model cascade is based. It is called directly from the basemodel very early in the model initialisation phase when the MPI environment is set up:

  - First of all, the rank of the current process entity (PE) in the MPI world communicator (`m_world_rank`) and the "world wide" number of tasks (PEs) within this MPI environment (`m_world_npes`) are acquired from MPI.
  - Secondly, the tasks are associated to the individual models. The model with rank 0 reads the MMD namelist (call of subroutine `read_coupling_layout`, see below, and the introduction). Based on the namelist settings the MPI layout is calculated:
    - Following the order of models in the coupling setup, each model gets the number of required tasks, i.e., the model with coupling `Id=1` is attributed to the tasks with `world_rank` 0 up to the number of requested PEs-1. For the example given below (and in the introduction), ECHAM5 would be associated with the tasks of rank 0 to $NPE[1]-1, the COSMO model with Id=2 is associated to the tasks with rank $NPE[1] to $NPE[1]+$NPE[2]-1, and so forth.
    - Based on the layout of the MMD models, i.e., the number of coupled models and the number of tasks of each model, the lowest rank (in the world communicator) for each model (the `start_PE`) is calculated.
    - The number of coupled models (`m_NrOfCpl`) and the start PEs are broadcasted to all PEs.
    - `start_PE` is then used by each PE to determine the model ID within the overall coupling setup (`m_my_CPL_Id`).
    - The relative rank of each task (`m_my_CPL_rank`) within one group of tasks defined by one model is determind by the difference of the world rank of the PE (`m_world_rank`) and the `start_PE` of the respective model.
    - The two parameters (`m_my_CPL_rank` and `m_my_CPL_Id`) are used to split the `MPI_comm_world` communicator (by calling the MPI routine `MPI_Comm_split`), yielding the group communicator for the respective model (`m_model_comm`).
    - With the group communicator the rank (`m_model_rank`) of the current PE in the respective group (=model) is determined, and
    - the number of processes combined in the group (`m_model_npes`) is inquired.

    Figure 4 illustrates the definition of the above mentioned variables. Note: if not denoted otherwise, "PE number" always refers to the rank of the current PE in the model specific group communicator.
  - After setting up the basic communicators the contents of the namelist, i.e., the `name`, the `Id` and the `ParentId`, are broadcasted to all PEs. The meaning of these variables is explained in the example below.
  - Based on this information, the individual communicators for the direct communication between parent and child model(`m_to_child_comm`) and vice versa (`m_to_parent_comm`) are determined. As a Parent can feed a number of child models, `m_to_child_comm` is a 1-dimensional array with dimension `MMD_MAX_MODEL`.
  - For chils models, the parent model type (`m_ParentType`) is set depending on the parent model name. `m_ParentType` is an INTEGER and is set to `MMD_ParentIsECHAM`, if the parent name is 'echam' and to `MMD_ParentIsCOSMO`, if it is 'cosmo'. For future applications of the MMD library other PARAMETERs defining a parent model type can be added.
  - A parent model additionally needs a list of the Ids of its child models. This information is stored in the 1-dimensional array `MMD_Parent_for_Child`, which is allocated by the number of child models a respective parent has to deal with.

- `MMD_Print_Error_Message`:

`SUBROUTINE MMD_Print_Error_Message`		`(iu, MMD_status)`	
name	type	intent	description
mandatory arguments:			
iu	INTEGER	IN	unit for output
MMD_status	INTEGER	IN	status flag

This is a utility subroutine printing individual error messages for predefined error stati.

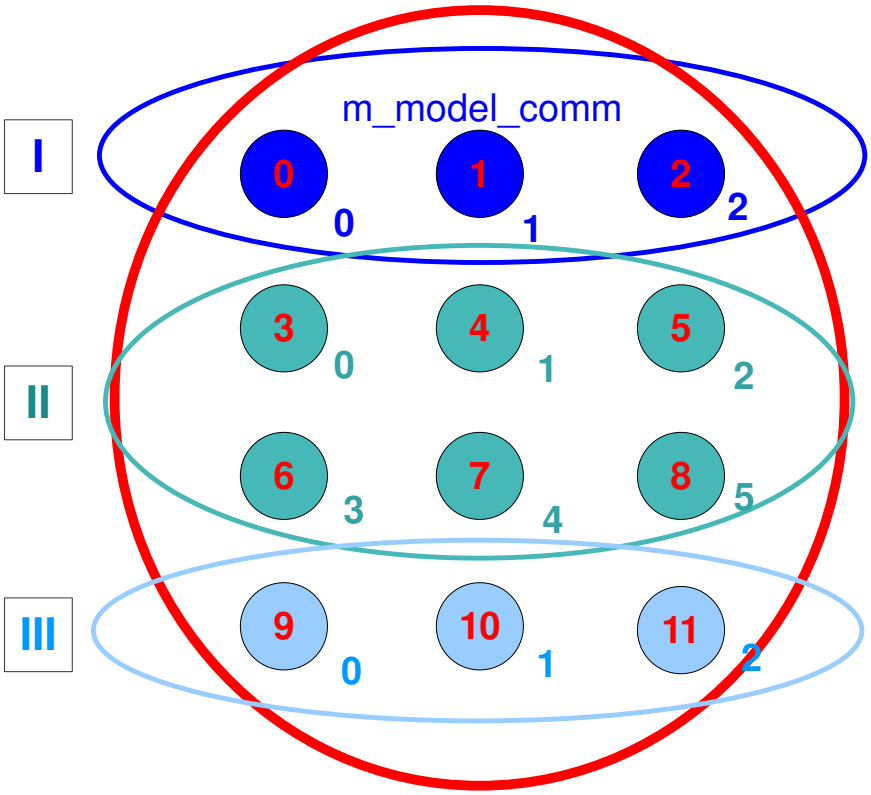


Figure 4: Illustration of the variables defined in `mmd_handle_communicator`: The small circles symbolise the individual tasks. The red circle encloses all tasks visualising the global MPI-communicator (`MPI_comm_world`). The overall number of tasks `m_world_npes` is 12 in this example. The rank of each individual task in the global communicator (`m_world_rank`) is indicated by the red numbers in the small circles. In this example the number of coupled models (`m_NrOfCpl`) is 3. Each of the larger bluish ellipses indicates one model group communicator (`m_model_comm`). For easier reference the models are indicated by roman numbers at the left hand side of the ellipses. The tasks are coloured identically to the ellipses. `m_model_npes` is 3 for the models I and III, whereas it is 6 for model number II. The number at the lower right of the small circles denotes the rank of the tasks in the model group communicators (`m_model_rank`). The `start_PE`s for the three models are the tasks with the MPI world rank 0, 3 and 9, respectively. `m_my_CPL_Id` is 1 for the tasks 0-2 (rank in the world communicator, red numbers) as they belong to model I. For model II `m_my_CPL_Id` is 2 and for modell III it is 3.

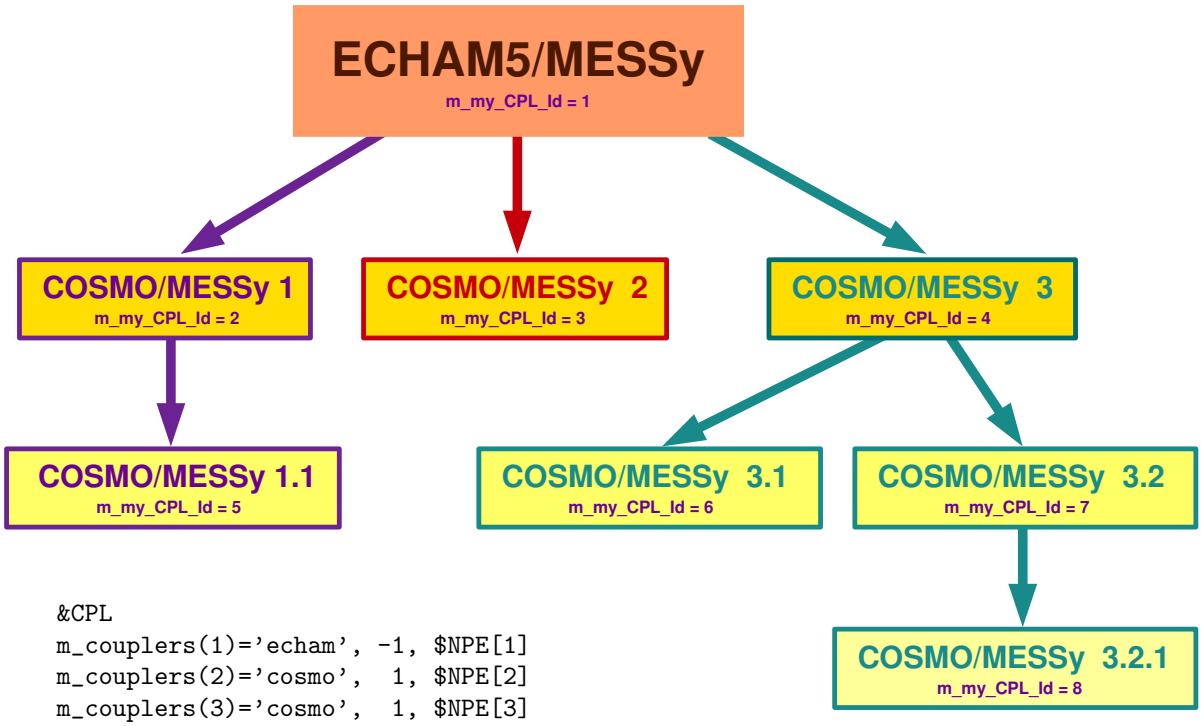
- `MMD_FreeMem_Communicator`:

`SUBROUTINE MMD_FreeMem_Communicator ()`

At the very end of the model integration allocated memory needs to be released. This subroutine deallocates the memory allocated for `MMD_Parent_for_Child`.

- `PRIVATE read_coupling_layout`:

`SUBROUTINE read_coupling_layout (MMD_status)`			
name	type	intent	description
mandatory arguments:			
MMD_status	INTEGER	INOUT	error/status flag




```
&CPL
m_couplers(1)='echam', -1, $NPE[1]
m_couplers(2)='cosmo',  1, $NPE[2]
m_couplers(3)='cosmo',  1, $NPE[3]
m_couplers(4)='cosmo',  1, $NPE[4]
m_couplers(5)='cosmo',  2, $NPE[6]
m_couplers(6)='cosmo',  4, $NPE[7]
m_couplers(7)='cosmo',  4, $NPE[8]
m_couplers(8)='cosmo',  7, $NPE[9]
/
```

Figure 5: Example for a possible MMD model layout.

The subroutine `read_coupling_layout` is called from the MMD subroutine `MMD_cag_model_communicator` as the coupling layout must be known for the communicator setup. The layout is determined by the user within the MMD library namelist file `MMD_layout.nml`. The required information is:

- the number of tasks associated to each individual model,
- the type of the model (currently 'echam' for ECHAM5/MESSy or 'cosmo' for COSMO/MESSy), and
- the parent ID of each model.

An example is illustrated in Fig. 5 with the global chemistry climate model ECHAM5/MESSy as *patriarch* und the regional COSMO/MESSy model as child models.

Each line defines one model within the MMD setup. `m_couplers` is the variable name of the structure containing the setup information within MMD. The index is the model instance, i.e., the MMD internal number or its `Id`. It must be unique for each model. The first column gives the name of the basemodel. The second column contains the ID of the parent of the respective model (the so-called `ParentId`). `-1` in the entry for 'echam' signifies that this model has no parent model, in other words it is the *patriarch*. The third column determines the number of Process Entities (PEs) required for each model[8], these are usually set by the run-script. In the example in Fig. 5, ECHAM5/MESSy (i.e., `'echam'` in the namelist) is defined as *patriarch*. It is Parent for the three COSMO/MESSy models with the MMD internal Ids

[8]In the usual ECHAM5/MESSy setup this is equivalent to `NCPUS= NPROCA` × `NPROCB`, whereas for the COSMO model this is the product of `nprocx` and `nprocy`.

2,3,4. The MMD internal model number 5 is Child of the model with Id 2, i.e., "COSMO/MESSy 1" in our example (Fig. 5). The MMD internal models number 6 and 7 are Children of the model number 4 ("COSMO/MESSy 3" in Fig. 5). Last but not least, the model with the internal number 8 is Child of the model with Id 7 ("COSMO/MESSy 3.2").

At the end of the subroutine `read_coupling_layout`, after reading the namelist file, the number of coupled models (`m_NrOfCpl`) is determined according to the namelist settings.

3.1.3 `mmd_mpi_wrapper`

The module `mmd_mpi_wrapper` provides six high level interface routines for data exchange between child and parent models and vice versa. Send and Receive for child and parent model are managed by four subroutines:

- `MMD_Send_to_Parent`:

`SUBROUTINE MMD_Send_to_Parent`		`(buf, n, Parent_rank, tag, ierr)`	
name	type	intent	description
mandatory arguments:			
buf	INTEGER, DIMENSION(:)	INOUT	1D integer data array to send
	INTEGER, DIMENSION(:,:)	INOUT	2D integer data array to send
	REAL(dp), DIMENSION(:)	INOUT	1D real data array to send
	REAL(dp), DIMENSION(:,:)	INOUT	2D real data array to send
	REAL(dp), DIMENSION(:,:,:)	INOUT	3D real data array to send
n	INTEGER	IN	length of buffer
Parent_rank	INTEGER	IN	rank of receiving Parent PE in the Parent group MPI-communicator
tag	INTEGER	IN	tag for data transfer to unambiguously identify this data package
ierr	INTEGER	OUT	error flag

This subroutine is called by a child model and sends a data array from the child to the parent model.

- `MMD_Recv_from_Child`:

`SUBROUTINE MMD_Recv_from_Child`		`(Child_Id, buf, n, Child_rank , tag, ierr)`	
name	type	intent	description
mandatory arguments:			
Child_Id	INTEGER	IN	ID of sending Child model
buf	INTEGER, DIMENSION(:)	INOUT	1D integer data array to receive
	INTEGER, DIMENSION(:,:)	INOUT	2D integer data array to receive
	REAL(dp), DIMENSION(:)	INOUT	1D real data array to receive
	REAL(dp), DIMENSION(:,:)	INOUT	2D real data array to receive
	REAL(dp), DIMENSION(:,:,:)	INOUT	3D real data array to receive
n	INTEGER	IN	length of buffer
Child_rank	INTEGER	IN	rank of sending Child PE in the Child group MPI-communicator
tag	INTEGER	IN	tag for data transfer to unambiguously identify this data package
ierr	INTEGER	OUT	error flag

This subroutine is called by the parent model and receives a data array from the child model.

- `MMD_Recv_from_Parent`:

`SUBROUTINE MMD_Recv_from_Parent`		`(buf, n, Parent_rank, tag, ierr)`	
name	type	intent	description
mandatory arguments:			
buf	INTEGER, DIMENSION(:)	INOUT	1D integer data array to receive
	INTEGER, DIMENSION(:,:)	INOUT	2D integer data array to receive
	REAL(dp), DIMENSION(:)	INOUT	1D real data array to receive
	REAL(dp), DIMENSION(:,:)	INOUT	2D real data array to receive
	REAL(dp), DIMENSION(:,:,:)	INOUT	3D real data array to receive
n	INTEGER	IN	length of buffer
Parent_rank	INTEGER	IN	rank of sending Parent PE in the MPI group communicator for the Parent
tag	INTEGER	IN	tag for data transfer to unambiguously identify this data package
ierr	INTEGER	OUT	error flag

This subroutine is called by the child model and receives a data array from the parent model.

- `MMD_Send_to_Child`:

`SUBROUTINE MMD_Send_to_Child`		`(Child_Id, buf, n, Child_rank , tag, ierr)`	
name	type	intent	description
mandatory arguments:			
Child_Id	INTEGER	IN	ID of Child model (*Receiver* of the data)
buf	INTEGER, DIMENSION(:)	INOUT	1D integer data array to send
	INTEGER, DIMENSION(:,:)	INOUT	2D integer data array to send
	REAL(dp), DIMENSION(:)	INOUT	1D real data array to send
	REAL(dp), DIMENSION(:,:)	INOUT	2D real data array to send
	REAL(dp), DIMENSION(:,:,:)	INOUT	3D real data array to send
n	INTEGER	IN	length of buffer
Child_rank	INTEGER	IN	rank of receiving Child PE in the MPI group communicater for the Child model (Mostly rank = 0 is chosen.)
tag	INTEGER	IN	tag for data transfer to unambiguously identify this data package
ierr	INTEGER	OUT	error flag

This subroutine is called by the parent and sends a data array from the parent to the child model.

With these routines the user does not have to take care about the internal MPI setup and the communicators. Moreover, the routines are overloaded for different data types: 1D-`integer` arrays, 2D-`integer` arrays and 1D-, 2D- and 3D-`real` arrays, respectively.

In addition, two broadcasting subroutines have been implemented:

- `MMD_Inter_Bcast`:

`SUBROUTINE MMD_Inter_Bcast`		`(buf, sender [, Child_Id] [, ierr])`	
name	type	intent	description
mandatory arguments:			
buf	INTEGER, DIMENSION(:)	INOUT	INTEGER buffer to exchange
buf	CHARACTER(LEN=*)	INOUT	CHARACTER buffer to exchange
optional arguments:			
Child_Id	INTEGER	IN	ID of (*remote*) Child (used by Parent only).
ierr	INTEGER	OUT	error flag

This subroutine is called `MMD_Inter_Bcast` and it provides the possibility to exchange 1-dimensional INTEGER or CHARACTER(LEN=*) arrays between *remote models*. The broadcasting subroutines can be easily overloaded to handle additional data types, so far there was no need to do so.

- `MMD_Bcast`:

`SUBROUTINE MMD_Bcast`		`(buf, root_pe [, comm] [, ierr])`	
name	type	intent	description
mandatory arguments:			
buf	INTEGER	INOUT	INTEGER buffer to be broadcasted
	CHARACTER(LEN=*)	INOUT	CHARACTER buffer to be broadcasted
root_pe	INTEGER	IN	rank of the PE sending the data in the respective MPI group communicator
optional arguments:			
comm	INTEGER	IN	communicator
ierr	INTEGER	OUT	error flag

The subroutine `MMD_Bcast` can be used to broadcast CHARACTER strings or INTEGER values using an arbitrary communicator. Thus, it can be used to broadcast data to PEs of the same model (using the intra-model MPI-communicator, which is the default), to PEs of the *remote model* (set `comm` in the parameter list to the communicator of the *remote model* (i.e., `m_to_child_comm` or `m_to_parent_comm`, respectively), to both models of a Child-Parent Pair (use the inter-communicator) or all models (set `comm` to the world communicator).

3.1.4 `mmd_child.f90`

The Multi-Model-Driver (MMD) consists of two parts. The MMD library, which is discussed here, and one MESSy submodel MMD2WAY consisting of the sub-submodels MMD2WAY_CHILD and MMD2WAY_PARENT dealing with child and parent model side, respectively. The routines included in the MMD library module `mmd_child` contain the child model specific part of the MMD library and interact directly with the MESSy sub-submodel MMD2WAY_CHILD. `mmd_child` includes thirteen subroutines and three functions:

- `MMD_C_GetParentType`:

`INTEGER FUNCTION MMD_C_GetParentType`	`()`

The INTEGER function `MMD_C_GetParentType` provides information about the associated parent model to the child MMD-subsubmodel. It returns an INTEGER value indicating the type of the parent model. At the time being this is one of `MMD_ParentIsECHAM` or `MMD_ParentIsCOSMO`.

- `MMD_C_Init`:

`SUBROUTINE MMD_C_Init`	`(l2way)`		
name	type	intent	description
optional arguments:			
l2way	LOGICAL	IN	indicates if two-way coupling is requested

At the beginning of the initialisation phase the child model side of the MMD library needs to be initialised, which is done in the subroutine `MMD_C_Init`. The variable `Me` of TYPE `ExchDataDef` is declared in the `mmd_child` module and provides all information required for the data exchange.

By calling the respective C-subroutine `MMDc_C_Init` the number of PEs occupied by the *remote model* is evaluated. The structure component `Me%PEs` and the MMD internal variable `BufLen`, which stores the length of the buffer received from each parent PE, are allocated according to the number of *remote PEs*. All POINTERs, which are not yet ASSOCIATEd are NULLIF(Y)ied, i.e., `Me%Ar`, `Me%ArrayStart`

and Me%PEs(:)%locInd are NULLIF(Y)ied. All structure components Me%PEs(:)%NrEle and the variable BufLen(:) are initialised with zero.

If 2-way coupling is employed, the same initialisation steps are performed with the variable Parent containing the information for the (back-)coupling to the Parent.

Additionally, the information whether this child model couples back to its parent model is stored in a module wide variable ltwoway. This information is provided by the optional argument l2way. If it is not present, 1-way coupling is assumed (ltwoway=.FALSE.).

- MMD_C_Set_DataArray_Name:

SUBROUTINE MMD_C_Set_DataArray_Name			(par_channel, par_object, chld_channel, chld_object, chld_repr, istat)
name	type	intent	description
mandatory arguments:			
par_channel	CHARACTER(LEN=*)	IN	name of Parent *channel*
par_object	CHARACTER(LEN=*)	IN	name of Parent *channel object*
chld_channel	CHARACTER(LEN=*)	IN	name of Child *channel*
chld_object	CHARACTER(LEN=*)	IN	name of Child *channel object*
chld_repr	CHARACTER(LEN=*)	IN	*representation* of Child *channel object* as given in the namelist
istat	INTEGER	OUT	error/status flag

The list of *exchange fields*, which is determined by the sub-submodel MMD2WAY_CHILD, needs to be initialised within the MMD library. MMD2WAY_CHILD reads a namelist containing a list of *exchange fields*, i.e., a list of those files required by the child model from the parent model. The subroutine MMD_C_Set_DataArray_Name builds a concatenated list of these fields. The structure component Me%ArrayStart points to the memory of the first *exchange field*, whereas all data arrays are stored in the concatenated list Me%Ar. The *channel* and *channel object* names of the *exchange fields* are stored in the structure components Me%Ar%Arrdef%channel and Me%Ar%Arrdef%object, respectively. Additionally, the parent *channel* and *channel object* names, the child *representation* as given in the MMD2WAY_CHILD namelist file, and an index are broadcasted to the parent model.

- MMD_C_Set_DataArray_EndList:

SUBROUTINE MMD_C_Set_DataArray_EndList	()

When all data fields are initialised, the end of the list is indicated by calling the subroutine MMD_C_Set_DataArray_EndList. In this subroutine the coupling index is set to -1. This value is interpreted as list end on the parent model side of the library.

- MMD_C_Get_ParDataArray_Name:

SUBROUTINE MMD_C_Get_ParDataArray_Name			(numfields)
name	type	intent	description
mandatory arguments:			
numfields	INTEGER	OUT	number of fields requested by the Parent

For the coupling of the data fields from the child to the parent model, the *channel* and *channel object* names of the *exchange fields* on the child model side, the *representation* name as provided by the parent model (given as namelist parameter), the required interpolation method, and a logical flag, indicating if the unit of the requested field is required by the parent, must be received by the child from its parent model. Within the subroutine MMD_C_Get_ParDataArray_Name these information are acquired from the parent model by MMD_Bcasts. First the couple_index is received. A value of -1 indicates the end of the transmission of the list and the subroutine is exited. Based on the received data the concatenated list (part of the ExchDataDef structure) defining the individual data fields is established. The memory location of the first array is stored in the variable Parent%ArrayStart. The

child *channel* and *channel object* names and the parent *representation* name are stored in the structure components `Parent%Ar%ArrDef%channel`, `Parent%Ar%ArrDef%object` and `Parent%Ar%ArrDef%repr`, respectively, making these strings available for later use on the parent part of the MMD library. Additionally, the structure components `Parent%Ar%ArrDef%interpM` and `Parent%Ar%ArrDef%l_sentunit` store the requested interpolation method and the request for the fields unit, respectively.

- `MMD_C_GetNextParArray`:

`LOGICAL FUNCTION MMD_C_GetNextParArray (MyChannel, myName, repr, interpM, l_SentUnit)`			
name	type	intent	description
mandatory arguments:			
MyChannel	CHARACTER(LEN=*)	OUT	name of *channel*
myName	CHARACTER(LEN=*)	OUT	name of *channel object*
repr	CHARACTER(LEN=*)	OUT	parent representation of the field
interpM	INTEGER	OUT	interpolation method
l_SentUnit	LOGICAL	OUT	sent unit of field to parent

After the call of the subroutine `MMD_C_Get_ParDataArray_Name` the information about the field requested by the parent model are available in a concatenated list. The function `MMD_C_GetNextParArray` makes this information available. With each call of this function the child model steps along the concatenated list and provides the information of the current list element. These are the *channel* and *channel object* name of the field in the child model, the representation of the field in the parent model, the requested interpolation method, and the logical indicating if the parent requires the unit of the field in the child model.

As long as new elements are available in the concatenated list, `MMD_C_GetNextParArray` is `.TRUE.`. If the end of the list is reached, `MMD_C_GetNextParArray` is set to `.FALSE.`.

- `MMD_C_Set_ParIndexlist`:

`SUBROUTINE MMD_C_Set_ParIndexlist (index_list, fractions, wfunc)`			
name	type	intent	description
mandatory arguments:			
index_list	INTEGER, DIMENSION(:,:)	INOUT	index list (used for the horizontal element association) as calculated from the Child submodel
fractions	REAL, DIMENSION(:,:)	IN	
wfunc	REAL, DIMENSION(:,:)	IN, OPTIONAL	

The most tricky part of the MMD library is the association of grid points from the parallel decomposed child "out"-field[9] (the field to sent) with the grid points on the parallel decomposed Parent fields (the received field).

For that, the child submodel MMD2WAY_CHILD sends the geographical longitude and the geographical latitude fields of the parallel decomposed parent *out-grid*[10], each as three dimensional fields: The first two ranks spread the horizontal distribution of the geographical longitude or latitude fields, respectively, as defined on one PE, the third rank is the respective PE number in the model specific MPI-group-communicator, i.e., the blue numbers at the lower right side of the PEs in Fig. 4. These fields contain for each index triple (i_c,j_c, PE_c) the geographical coordinates. Based on the geographical coordinates, the parent submodel MMD2WAY_PARENT identifies for each of the child "out"-grid points the respective grid point in the local parent model domain, thus adding to the list the parent process number PE_p (of the model specific MPI-communicator) on which the respective geographical point is located and the respective

[9] As the interpolation from the child model to the parent grid is performed by the child model, the data fields sent by the child model are already on the parent grid. This field will be called "out"-field in the following. In case of MECO(n) the grid of the "out"-field is the same as the grid of the "in"-field.

[10] As the *in-* and *out-grid* can be rotated grids, the geographical longitude and latitude fields are 2D fields each.

index pair (i_p,j_p) in the local domain. Thus a list of n sextuples (i_p,j_p,i_c,j_c,PE_c, PE_p)$_n$ containing the index pairs of both decomposed fields and the child and the parent PE number is created, with n being the overall number of exchanged horizontal elements.

This list is sent to MMD2WAY_CHILD which forwards it as parameter to the MMD library subroutine `MMD_C_Set_ParIndexlist`, where it is further analysed to yield all the information required for a most efficient data exchange. Additionally, a 2D-field (`fraction`) in "out"-grid geometry is parameter to the subroutine, indicating which fraction of each grid box of the "out"-grid is actually covered by the local child domain. The third parameter of the subroutine is the weight function (`wfunc`) containing the weight which should be applied when changing the original parent field. This weighting function is 1 (100%) in most of the domain and diminishing to 0 at the border of the child model domain.

At the beginning, the task of `m_model_rank=0` sorts the index list by the child model PE numbers and calculates the number of grid points each child PE has to sent (`Parent%NrPoints`). This number is sent to the respective child PE. Additionally, the part of the index list of the respective child PE is sent to it. Each child PE deduces from its part of the index list the number of horizontal elements `Parent%PEs(ip)%NrEle` it has to send to each individual parent PE. After that, the index pairs of the local child grid associated with the horizontal elements are saved in the `locInd` structure `Parent%PEs(ip)%locInd`. Additionally, the index pairs of the local parent grid, the fraction and the weight function of the respective element are saved in the respective components of the local structure variable `par_ind`. In the following the number of horizontal elements and the components of the local variable `par_ind` are exchanged with the respective parent PEs.

Note: As the buffer exchange is based on the index list as explained above, the buffer exchange can only be used for fields which contain both horizontal dimensions. If other fields should be exchanged during a simulation, this has to be performed via the subroutines provided by the module `mmd_mpi_wrapper` (Sect. 3.1.3).

- `MMD_C_Set_ParDataArray`:

SUBROUTINE MMD_C_Set_ParDataArray		(status, DimLen, ArrayOrder, p4)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	status flag
DimLen	INTEGER,DIMENSION(4)	IN	length of the 4 dimensions
ArrayOrder	CHARACTER(LEN=4)	IN	*axis string* indicating order of axes
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	POINTER for 4D data arrays

The subroutine `MMD_C_Set_ParDataArray` is called for each of the fields individually. It associates the respective POINTER of the array and stores

  - the dimension lengths (`Parent%Ar%ArrDef%dim`),
  - the order of dimensions (`Parent%Ar%ArrDef%dim_order`), and
  - calculates the axis indices `Parent%Ar%ArrDef%xyzn_dim`.

Additionally,

  - the array length (`Parent%Ar%Arrdef%ArrLen(ip)`),
  - the array index (`Parent%Ar%Arrdef%ArrIdx(ip)`), and
  - the buffer length (`ParBufLen(ip)`)

are calculated from the above information. Subsect. 3.3 illustrates the meaning of these variables.

- `MMD_C_Get_Indexlist`:

SUBROUTINE MMD_C_Get_Indexlist ()

The most important contribution of the parent model to the data exchange, apart from the data itself, is the list attributing the data points of the parallel decomposed parent grid to the parallel decomposed

child *in-grid*. The parent calculates the index list, which interlinks the data grid point (i_p,j_p) on parent process PE_p with the child process PE_c and the local *in-field* grid box (i_c,j_c) (see description to Fig. 3, Table 1 and Sect. 3.1.5). The subroutine `MMD_C_Get_Indexlist` processes the data made available by the parent model. Each parent PE sends the number of elements `NrEle`, which will be sent during the buffer exchange to the respective child PE. Accordingly, this information is stored in the structure component `Me%PEs(ip)%NrEle`. Subsequently, the index pairs associated to the elements are sent from the parent and stored in `Me%PEs(ip)%locInd`. Here, `ip` is the number of the respective sending parent PE.

- `MMD_C_Get_Repr`:

`SUBROUTINE MMD_C_Get_Repr`		`(axis, gdimlen, name, att)`	
name	type	intent	description
mandatory arguments:			
axis	CHARACTER(LEN=4)	OUT	string indicating axes order
gdimlen	INTEGER, DIMENSION(4)	OUT	length of dimensions
name	CHARACTER(LEN=STRLEN_CHANNEL)	OUT	*representation* name
att	CHARACTER(LEN=STRLEN_ULONG)	OUT	*channel object attribute* (e.g. height axis)

After the internal setup of the MMD library, the memory for the *in-fields* needs to be initialised within the child MESSy submodel MMD2WAY_CHILD and the POINTER to this memory is handed to the MMD library. The allocation of the memory within MMD2WAY_CHILD is described in detail within the "MMD user manual"[11]. In order to enable the exchange of fields, which *representation* is not a priori known, the subroutine `MMD_C_Get_Repr` (in `mmd_child`) and `MMD_P_Sent_Repr` (in `mmd_parent`) have been added to the MMD library to exchange the information required to determine the *representation* on the child model side from the information given by the parent model. These routines are only called, if the *representation* in the MMD2WAY_CHILD namelist file was set to `'#UNKNOWN'`. In this case the parent sends the *representation* name of the respective Parent *channel object*. Additionally,

- the *axis string*,
- the global *dimensions*, and
- an additional *attribute* (e.g., emission heights)

are exchanged. These parameters are made available on the Child side within the subroutine `MMD_C_Get_Repr` and handed to the MMD2WAY_CHILD submodel via parameter list.

- `MMD_C_GetNextArray`:

`LOGICAL FUNCTION MMD_C_GetNextArray`		`(MyChannel, myName)`	
name	type	intent	description
mandatory arguments:			
MyChannel	CHARACTER(LEN=*)	OUT	name of *channel*
myName	CHARACTER(LEN=*)	OUT	name of *channel object*

After the allocation of the memory required by the child model submodel, the respective POINTERs can be made available to the MMD library. For this the child model steps along the concatenated list provided by MMD with the help of the MMD function `MMD_C_GetNextArray`. This function provides the required *channel* and *channel object* name to the child submodel MMD2WAY_CHILD. With each call, this function internally steps one entry forward within the concatenated list.

[11] The MMD user manual is available in the same electronic supplement as this manual.

- `MMD_C_Set_DataArray`:

`SUBROUTINE MMD_C_Set_DataArray`		`(status, DIMLEN, ArrayOrder, p4)`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	status flag
DimLen	INTEGER,DIMENSION(4)	IN	length of the 4 dimensions
ArrayOrder	CHARACTER(LEN=4)	IN	*axis string* indicating order of axes
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	POINTER for 4D data arrays

The subroutine `MMD_C_Set_DataArray` is called for each of the fields individually. It associates the respective POINTER of the array and stores

  - the dimension lengths (`Me%Ar%ArrDef%dim`),
  - the order of dimensions (`Me%Ar%ArrDef%dim_order`), and
  - calculates the axis indices `Me%Ar%ArrDef%xyzn_dim`.

  Additionally,

  - the array length (`Me%Ar%Arrdef%ArrLen(ip)`),
  - the array index (`Me%Ar%Arrdef%ArrIdx(ip)`), and
  - the buffer length (`BufLen(ip)`)

  are calculated from the above information. Subsect. 3.3 illustrates the meaning of these variables.

- `MMD_C_SetInd_and_AllocMem`:

`SUBROUTINE MMD_C_SetInd_and_AllocMem`	`()`

When the definition of all data fields within the Fortran95 interface of the MMD library is complete, the subroutine `MMD_C_SetInd_and_AllocMem` must be called to initialise the total length of the buffers within the C-core of the library. The total buffer length corresponds to the memory that must be allocated for the buffer exchange (see Sect. 3.2.5). As the amount of the required memory is defined differently in the cases of 1-way or 2-way coupling, different C functions are called depending on the coupling type.

At this point, all preparations are complete and the data exchange can be performed.

- `MMD_C_GetBuffer`:

`SUBROUTINE MMD_C_GetBuffer`		`(WaitTime)`	
name	type	intent	description
optional arguments:			
WaitTime	REAL(dp)	OUT	time waiting until buffer is available

To actually exchange the data from parent to the child model during the integration phase, the parent writes the required data into the memory buffers made available by `MPI_alloc_mem`. To read these buffers, the subroutine `MMD_C_GetBuffer` calls, independently for each parent PE, its C counterpart `MMDc_C_GetBuffer` (see Fig. 2). The C function hands back a 1-dimensional array. This is transferred back into its full 4-dimensional structure within the Fortran95 part of the library. For this back transformation the indices for the different *dimensions* and the `dim_order` label are used (see Sect. 3.3). `MMD_C_GetBuffer` contains a generic routine for the back transition of the 1-dimensional arrays to all dimension orders. In addition, more computationally efficient implementations for the most often used *representations* are provided as special cases, e.g. the standard axis orders `'XY--'` and `'XYZ-'`. By transforming the fields sent by each parent PE into their usual 4D structure, the *in-fields* of the child submodel are filled and can be processed by the child submodel afterwards. During the integration, this subroutine can be called as often as required.

- SUBROUTINE MMD_C_FillBuffer

```
SUBROUTINE MMD_C_FillBuffer          ( )
```

If two-way coupling is ongoing, the subroutine MMD_C_FillBuffer fills the buffer within the time loop. The C function actually filling the buffer requires a 1-dimensional array as input. Thus the 4-dimensional data needs to be re-ordered. As the dimension order of the child arrays is not a priori known on the parent side, the packing algorithm in MMD_C_FillBuffer packs the arrays invariably in the same order:

- The loop over the elements in the xy-plane (NrEle) is the slowest,
- next is the loop over the 'Z' dimension, and
- fastest varying is the 'N' dimension.

The MMD_C_FillBuffer contains an algorithm packing an array of arbitrary order of *dimensions* in grid point *representation*[12]. For higher computing efficiency, the commonly used *dimension orders* are implemented as special cases (e.g. 'XY--', 'XZY-' or 'XYZN'). If required, other special cases can be included in this subroutine as well. For each 1-dimensional (i.e. packed) array per *remote PE* the C routine MMDc_C_FillBuffer is called, copying the array to the memory space allocated by MPI_Alloc_mem. When all buffers attributed to all parent PEs are filled, the C function MMDc_C_SetBarrier is called, which sets a barrier to prevent the buffer to be filled a second time before the data was read by the parent.

- MMD_C_FreeMem:

```
SUBROUTINE MMD_C_FreeMem                ()
```

After the integration, the memory allocated during the initialisation phase is deallocated. This is done within the subroutine MMD_C_FreeMem.

3.1.5 mmd_parent.f90

The module mmd_parent contains the parent model specific Fortran95 part of the MMD library. In contrast to the child model side, the parent can provide data to more than one child model. Therefore, many of the structures used in the child submodel require an array dimension for the parent side. In the following, two indices to identify a specific child model are distinguished:

- The variable ChildId always indicates the index of the child model in the entire MMD setup. This is dimensioned by the PARAMETER MMD_MAX_MODEL, which is set to 64 at the moment. The association of the ChildId depends on the entries in the MMD namelist file MMD_layout.nml. In mmd_parent this index is used to address the correct child model communicator (m_to_child_comm), and the index is parameter of almost all calls of the subroutines of the C part of the MMD library, because the C part of the library uses exclusively this index (see Sect. 3.2).
- After the initial setup of MMD the number of child models for each individual parent is known. Thus, the data definition structure (and other variables the parent has to define for each child model separately) are dimensioned by the number of child models of the respective parent. Id is the index used to address the child specific data structure of the specific parent.

Hereafter, the subroutines provided by the module mmd_parent are listed. All routines, except MMD_P_Allocate_Child and MMD_P_FreeMem are called separately for each child model. Thus one of the two indices introduced above (ChildId or Id) is always parameter of the subroutine calls to identify the respective child model.

[12]It is presumed that the array is defined in the horizontal space as the buffer exchange only works in this case.

- `MMD_P_Allocate_Child`:

`SUBROUTINE MMD_P_Allocate_Child`			`(NumChildren, l2way)`
name	type	intent	description
mandatory arguments:			
NumChildren	INTEGER	IN	number of Child models
l2way	LOGICAL	IN	.TRUE. if backward coupling of Child models is required

At the very beginning, in the subroutine `MMD_P_Allocate_Child` a pointer of TYPE `ExchDataDef` (named `Child`) is allocated according to the number of child models served by the specific parent model. Additionally, the variable for the length of the exchange buffer (`ChldBL`) to the respective child model needs to be dimensioned also by the number of child models. If `l2way == .TRUE.`, coupling back of at least some child model is required, and the structure `Parent` of TYPE `ExchDataDef` needs to be allocated to the number of child models (not only those which are coupled back). The buffer length variable `ParBL` is allocated accordingly. Furthermore, the structure components, which are pointers, are NULLIF(Y)ied.

Last but not least, the information, if the simulation is a 1-way or a 2-way coupled simulation is saved in a module wide variable (`ltwoway =l2way`).

- `MMD_P_Init`:
This subroutine is called separately for each child of a parent.

`SUBROUTINE MMD_P_Init`			`(ChildId, Id)`
name	type	intent	description
mandatory arguments:			
ChildId	INTEGER	IN	index of child model within the overall MMD model setup
Id	INTEGER	IN	index of child model in the child list of this specific parent

First, the parent needs to be initialised for each child model. The subroutine `MMD_P_Init` inquires the number of processes occupied by the respective child model (`Child(Id)%inter_npes`) by calling the MMD subroutine `MMDc_P_Init` and allocates `Child(Id)%PEs` and `ChldBL(Id)%BufLen`, accordingly. `Child(Id)%PEs(ip)%NrEle` and `ChldBL(Id)%BufLen(ip)` are initialised with 0 (`ip` is the index of the *remote PE*) and `Child(Id)%PEs(ip)%locInd` is NULLIF(Y)ied.

In case of 2-way coupling, the information acquired for `Child(Id)` are also saved in the respective `Parent(Id)` structure components, i.e. `Parent(Id)%ChildId` and `Parent(Id)%inter_npes` are set. The structure components of `Parent(Id)` are preset in the same way as those of `Child(Id)`.

- `MMD_P_Set_Indexlist`:

`SUBROUTINE MMD_P_Set_Indexlist`			`(Id, index_list)`
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child model list of this specific parent
index_list	INTEGER, DIMENSION(:,:)	INOUT	index list (used for the horizontal element association) as calculated by the parent submodel

The most tricky part of the MMD library is the association of grid points of the parallel decomposed parent grid with the grid points of the parallel decomposed child model *in-grid*. For that, the parent submodel MMD2WAY_PARENT receives the geographical longitude and the geographical latitude fields of the parallel decomposed child "in"-grid[13], each as three dimensional fields: The first two ranks spread the horizontal distribution of the geographical longitude or latitude fields, respectively, as defined on one PE, the third rank is the respective PE number in the model specific MPI-group-communicator, i.e., the

[13]As the parent and the child model "in"-grids can be rotated grids, the geographical longitude and latitude fields are 2D fields each.

blue numbers at the lower right side of the PEs in Fig. 4. These fields contain for each index triple (i_c,j_c, PE_c) the geographical coordinates. Based on the geographical coordinates, the parent submodel identifies for each of the child grid points the respective grid point in the local parent model domain, thus adding to the list the parent process number PE_p (of the model specific MPI-communicator) on which the respective geographical point is located and the respective index pair (i_p,j_p) in the local domain. Thus a list of n sextuples (i_p,j_p,i_c,j_c,PE_c, PE_p)$_n$ containing the index pairs in both decomposed fields, and the child and the parent PE number is created, with n being the overall number of exchanged horizontal elements.

This list is further analysed within the MMD library routine `MMD_P_Set_Indexlist` to yield all the information required for a most efficient data exchange. At the beginning, the task of `m_model_rank=0` sorts the index list by the parent PE numbers and calculates the number of grid points each parent PE has to sent (`Child(Id)%NrPoints`). This number is sent to the respective parent PE. Additionally, the part of the index list of the respective parent PE is sent to it. Each parent PE deduces from its part of the index list the number of horizontal elements `Child(Id)%PEs(ip)%NrEle` it has to send to each individual child PE. After that, the index pairs of the local parent grid associated with the horizontal elements are saved in the `locInd` structure `Child(Id)%PEs(ip)%locInd`, and the index pairs of the local Child grid are saved in an intermediate variable. Next, the number of horizontal elements and the intermediate variables are exchanged with the respective child PEs.

Note: As the buffer exchange is based on the index list as explained above, the buffer exchange can only be used for fields, which contain both horizontal dimensions. If other fields should be exchanged during a simulation, this has to be performed via the subroutines provided by the module `mmd_mpi_wrapper` (Sect. 3.1.3).

- `MMD_P_Get_ParIndexlist`:

`SUBROUTINE MMD_P_Get_ParIndexlist (Id, fractions, wfunc)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent
fractions	REAL(dp)	POINTER	grid box wise fractional overlap by the respective child model PE
wfunc	REAL(dp)	POINTER	weight function

This subroutine establishes the `Indexlist` for the coupling of the child model data to the parent. First, from each individual child model PE the number of elements provided by this PE is received by the parent. This number is stored in the structure component `Parent(Id)%PEs(ip)%NrEle`. Subsequently, `Parent(Id)%PEs(ip)%locInd` and two local variables are allocated accordingly. The latter are used to receive the information about the local index pair (`(i,j)`), the fractions and the weight functions. The index pair is stored in the structure component `Parent(Id)%PEs(ip)%locInd`, while the fraction and the weight function are copied to the subroutine parameter `fraction` and `wfunc`.

- `MMD_P_Set_ParDataArray_Name`:

`SUBROUTINE MMD_P_Set_ParDataArray_Name`			`(par_channel, par_object, chld_channel, chld_object, chld_repr, interpol_method, sentunit, istat)`
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	child model Id
par_channel	CHARACTER(LEN=*)	IN	name of parent *channel*
par_object	CHARACTER(LEN=*)	IN	name of parent *channel object*
chld_channel	CHARACTER(LEN=*)	IN	name of child *channel*
chld_object	CHARACTER(LEN=*)	IN	name of child *channel object*
chld_repr	CHARACTER(LEN=*)	IN	*representation* of child *channel object* as given in the namelist
interpol_method	INTEGER	IN	indicator for interpolation method used for interpolation between child and parent *representation* for this field
sentunit	LOGICAL	IN	Is the unit of the child field required by parent?
istat	INTEGER	OUT	error/status flag

The list of *exchange fields* requested by the parent, which is determined by the sub-submodel MMD2WAY_PARENT, needs to be initialised within the MMD library. MMD2WAY_PARENT reads a namelist containing a list of *exchange fields*, i.e., a list of those files required by the parent model from the respective child model. The subroutine `MMD_P_Set_ParDataArray_Name` is called seperately for each of these fields and builds a concatenated list of pointers to these fields. The structure component `Parent(Id)%ArrayStart` points to the memory of the first *exchange field*, whereas all data arrays are stored in the concatenated list `Parent(Id)%Ar`. The *channel* and *channel object* names of the *exchange fields* are stored in the structure components `Parent(Id)%Ar%Arrdef%channel` and `Parent(Id)%Ar%Arrdef%object`, respectively. Additionally, the *channel* and *channel object* name of these fields in the child model, their *representation*s -as given in the MMD2WAY_PARENT namelist file-, the interpolation methods, and the indicators, signifying if the field units need to be sent to the parent, are broadcasted to the child model.

- `MMD_P_Set_ParDataArray_EndList`:

`SUBROUTINE MMD_P_Set_ParDataArray_EndList`			`(Id)`
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model

When all data fields are initialised, the end of the list is indicated by calling the subroutine `MMD_P_Set_ParDataArray_EndList`. In this subroutine the coupling index is set to `-1`. This value is interpreted as list end from the child model side of the library.

- `MMD_P_Get_DataArray_Name`:

`SUBROUTINE MMD_P_Get_DataArray_Name`			`(Id)`
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model

For the coupling of the data fields required by the child model, the *channel* and *channel object* names of the *exchange fields* and the *representation* name listed in the namelist of the child model must be

received by the parent from each child model. Within the subroutine `MMD_P_Get_DataArray_Name` the *channel* and *channel object* names and the *representation* name are acquired from the child model by `MMD_Bcast`s. First the `couple_index` is received. A value of `-1` indicates the end of the transmission of the list and the subroutine is exited. Based on the received data, the concatenated list (part of the `ExchDataDef` structure) defining the individual data fields is established. The memory location of the first array is stored in the variable `Child(Id)%ArrayStart`. The parent *channel* and *channel object* name and the child *representation* name are stored in the structure components `Child(Id)%Ar%ArrDef%channel`, `Child(Id)%Ar%ArrDef%object` and `Child(Id)%Ar%ArrDef%repr`, respectively, making these strings available for later use on the parent model side of the MMD library.

- `MMD_P_GetNextArray`:

`LOGICAL FUNCTION MMD_P_GetNextArray (Id, MyChannel, myName, repr)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
MyChannel	CHARACTER(LEN=*)	OUT	parent *channel* name of data field
myName	CHARACTER(LEN=*)	OUT	parent *channel object* name of data field
repr	CHARACTER(LEN=*)	OUT	*representation* of data field (as given by the child model)

So far, the *channel* and *channel object* names as requested by the child model are only known within the MMD library itself. For the acquisition of the parent model fields, they must be made available for the parent submodel MMD2WAY_PARENT. With each call to the function `MMD_P_GetNextArray` the next element of the concatenated list is addressed, and the *channel* and *channel object* name and the *representation* name as provided by the child model are forwarded to the parent submodel MMD2WAY_PARENT. The parent submodel obtains the requested data POINTER based on the *channel* and *channel object* name by calling the subroutine `get_channel_object` of the MESSy infrastructure submodel CHANNEL.

- `MMD_P_Send_Repr`:

`SUBROUTINE MMD_P_Send_Repr (axis, gdimlen, name, att, ChildId)`			
name	type	intent	description
mandatory arguments:			
axis	CHARACTER(LEN=4)	INOUT	*axis string* of *representation*
gdimlen	INTEGER, DIMENSION(4)	INOUT	global *dimensions* of *representation*
name	CHARACTER(LEN=STRLEN_CHANNEL)	INOUT	name of *representation*
att	CHARACTER(LEN=STRLEN_ULONG)	INOUT	*attribute* of data object
ChildId	INTEGER	IN	index of the respective child model in the MMD model setup

If the *representation* name sent by the child model is `"#UNKOWN"`, the child requests further information about the *representation* of the respective *channel object* from the parent model (see description of the subroutine `MMD_C_Get_Repr` in Sect. 3.1.4). The MMD library routine `MMD_P_Send_Repr` sends the

- *axis string*,
- the global *dimensions*,
- the name of the *representation*, and
- an additional *attribute*

to the child model.

- `MMD_P_Set_DataArray`:

`SUBROUTINE MMD_P_Set_DataArray (Id, status, DIMLEN, ArrayOrder, p4)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
status	INTEGER	OUT	error/status flag
DimLen	INTEGER, DIMENSION(4)	IN	dimension length of input array
ArrayOrder	CHARACTER(LEN=4)	IN	*axis string* for input array
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	4D POINTER to the memory of the *exchange field*

For the coupling of the data requested by the child model, the POINTER to the memory of the parent model *exchange fields* are passed to the MMD library by the subroutine `MMD_P_Set_DataArray` and saved in the structure component `Child(Id)%Ar%Arrdef%p4`. Additionally,

- the dimensions (`Child(Id)%Ar%Arrdef%dim`),
- the dimension order (`Child(Id)%Ar%Arrdef%dim_order`) and
- the index of the `'X'`, `'Y'`, `'Z'` and `'N'` axes (`Child(Id)%Ar%ArrDef%xyzn_dim`)

are stored within the `Child` structure. Making use of these information, for each child model PE (index `ip`) the individual array length (`Child(Id)%Ar%Arrdef%ArrLen(ip)`) and the length of the buffer sent by each parent model PE to each child PE are calculated (`ChldBL(Id)%BufLen(ip)`).

- `MMD_P_GetNextParArray`:

`LOGICAL FUNCTION MMD_P_GetNextParArray (Id, MyChannel, myName)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
MyChannel	CHARACTER(LEN=*)	OUT	name of *channel*
myName	CHARACTER(LEN=*)	OUT	name of *channel object*

After the allocation of the memory required by the parent submodel for the coupling of the data requested by the parent, the respective POINTERs can be made available to the MMD library. For this, the MMD function `MMD_P_GetNextParArray` steps along the concatenated list provided by MMD. Within the function, a library internal pointer is set to the current list member. Additionally, this function provides the required *channel* and *channel object* name to the parent submodel MMD2WAY_PARENT. With each call, this function internally steps one entry forward within the concatenated list.

- `MMD_P_Set_ParDataArray`:

`SUBROUTINE MMD_P_Set_ParDataArray (Id, status, DIMLEN, ArrayOrder, p4)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
status	INTEGER	OUT	status flag
DimLen	INTEGER,DIMENSION(4)	IN	length of the 4 dimensions
ArrayOrder	CHARACTER(LEN=4)	IN	*axis string* indicating order of axes
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	POINTER for 4D data arrays

The subroutine `MMD_P_Set_ParDataArray` is called for each of the fields individually, making use of the MMD internal pointers set by the function `MMD_P_GetNextParArray`. It associates the respective POINTER of the array and stores

– the dimension lengths (`Parent%Ar%ArrDef%dim`),
– the order of dimensions (`Parent%Ar%ArrDef%dim_order`), and
– the calculated the axis indices (in `Parent%Ar%ArrDef%xyzn_dim`).

Additionally,

– the array length (`Parent%Ar%Arrdef%ArrLen(ip)`),
– the array index (`Parent%Ar%Arrdef%ArrIdx(ip)`), and
– the buffer length (`ParBL(Id)%BufLen(ip)`)

are calculated from the above information. `ip` is the index of the respective child model PE. Subsect. 3.3 illustrates the meaning of these variables.

- `MMD_P_SetInd_and_AllocMem`:

`SUBROUTINE MMD_P_SetInd_and_AllocMem (Id)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model

After all POINTERs and dimension information are stored, the size of the buffer for the exchange with the *remote model* is determined. For a 1-way coupling, this buffer length is equal to the sum over all buffer lengths sent to the individual child PEs (`SUM(ChldBL(Id)%BufLen)`). If 2-way coupling is required, data is exchanged in both directions. Thus, the exchange buffer needs to be dimensioned as required by the larger of the two exchange data sets `MAX(SUM(ChldBL(Id)%BufLen),SUM(ParBL(Id)%BufLen))`. This is triggered by the call of the subroutine `MMDc_P_SetInd_and_Mem` or `MMDc_P_SetInd_and_Mem_2way`, respectively. As the full size of the buffer is only required in the C part of the library the Fortran95 subroutine `MMD_P_SetInd_and_AllocMem` simply calls the C function `MMDc_P_SetInd_and_Mem`. After processing this subroutine the initialisation phase is finished.

- `MMD_P_FillBuffer`:

`SUBROUTINE MMD_P_FillBuffer (Id [, WaitTime])`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
optional arguments:			
WaitTime	REAL(dp)	OUT	idle time in seconds waiting for buffer release

Within the time loop the subroutine `MMD_P_FillBuffer` fills the buffer. The C function actually filling the buffer requires a 1-dimensional array as input. Thus the 4-dimensional data needs to be re-ordered. As the dimension order of the parent arrays is not a priori known on the child model side, the packing algorithm in `MMD_P_FillBuffer` packs the arrays invariably in the same order:

– The loop over the elements in the xy-plane (`NrEle`) is the slowest,
– next is the loop over the `'Z'` dimension, and
– fastest varying is the `'N'` dimension.

The `MMD_P_FillBuffer` contains an algorithm packing an array of arbitrary order of *dimensions* in grid point *representation*[14]. For higher computing efficiency, the commonly used *dimension orders* are implemented as special cases (e.g. `'XY--'`, `'XZY-'` or `'XYZN'`). If required, other special cases can be included in this subroutine as well. For each 1-dimensional (i.e. packed) array per *remote PE* the C routine

[14]It is presumed that the array is defined in the horizontal space as the buffer exchange only works in this case.

MMDc_P_FillBuffer is called, copying the array to the memory space allocated by MPI_Alloc_mem. When all buffers attributed to all child model PEs are filled, the C function MMDc_P_SetBarrier is called, which sets a barrier to prevent the buffer to be filled a second time before the data was read by the child model.

During the integration, this subroutine can be called as often as required, but only in turns with the subroutines MMD_C_GetBuffer, MMD_C_FillBuffer and MMD_P_FillBuffer for the 2-way coupling, or in case of 1-way coupling in turns with MMD_C_GetBuffer.

- MMD_P_GetBuffer:

SUBROUTINE MMD_P_GetBuffer		(Id, WaitTime)	
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
optional arguments:			
WaitTime	REAL(dp)	OUT	time waiting until buffer is available

To actually exchange the data from the child to the parent model during the integration phase, the child model writes the required data to the memory buffers made available by MPI_alloc_mem. To read this buffers the subroutine MMD_P_GetBuffer calls independently for each child PE its C counterpart MMDc_P_GetBuffer (see Fig. 2). The C function hands back a 1-dimensional array. This is transferred back into its 4-dimensional structure within the Fortran95 part of the library. For this back transformation the indices for the different *dimensions* and the dim_order label are used (see Sect. 3.3). MMD_P_GetBuffer contains a generic routine for the back transition of the 1-dimensional arrays to all dimension orders. In addition, more computationally efficient implementations for the most often used *representations* are provided as special cases, e.g. the standard axis orders 'XY--' and 'XYZ-'. By transforming the fields sent by each child model PE into their 4D structure, the *in-fields* of the parent submodel are filled and can be processed by the parent submodel afterwards. During the integration, this subroutine can be called as often as required, but only in turns with the subroutines MMD_P_FillBuffer, MMD_C_GetBuffer and MMD_C_FillBuffer.

- MMD_P_FreeMem:

SUBROUTINE MMD_P_FreeMem		(NumChildren)	
name	type	intent	description
mandatory arguments:			
NumChildren	INTEGER	IN	number of child models of this specific parent model

At the end of the simulation the allocated memory is deallocated within the subroutine MMD_P_FreeMem.

3.1.6 mmd_test.f90

The most tricky parts of the data exchange are to match the packing algorithms of the parent and the child model and the creation of the index_list, in which the index pair of each horizontal element of the child model *in-field* on each child PE is associated with the index pair of the associated horizontal element in the local parent domain and the respective PE number (in the parent specific group communicator). Whether the exchange of a horizontal field is performed properly can be tested, by creating an artificial additional *exchange field*. The additional field on the parent model side is a three dimensional field spanned by the usual horizontal plane and a number axis of length 8 (N=8). The 'Z' dimension is allocated with 1. The following values are assigned to the variable: The horizontal fields for N=1 and N=2 contain the geographical longitude and latitude, respectively. The third to eighth entry are identical to the index_list. Table 3 lists all entries.

During the exchange procedure each horizonal grid point and the associated eight entries are assigned to a grid point of the *in-field* of the child model. Thus the *in-fields* longitude and the latitude of each grid point as defined by the child model can directly be compared with the longitude and the latitude in the exchanged parent

Table 3: Meaning of the eight number dimensions of the `test_array`.

N	field
1	Geographical longitude as defined for the Parent grid
2	Geographical latitude as defined for the Parent grid
3	First index in parallel decomposed Parent grid (i_p)
4	Second index in parallel decomposed Parent grid (j_p)
5	First index in parallel decomposed Child grid (i_c)
6	Second index in parallel decomposed Child grid (j_c)
7	Number of respective Child process (PE_c)
8	Number of respective Parent process (PE_p)

field, i.e., the first and second entry of the *exchanged field*. If these are not equal, the exchange procedure went wrong. Additionally, the index pair of the *in-field* can be compared to the child model index pair contained in the *exchanged field*. They also have to be identical. This test does not sufficiently prove the correctness of the exchange procedure, but it checks the two most error-prone parts of the data exchange.

To perform this test, the following subroutines are provided by the module `mmd_test`. `mmd_test` is used by the parent and child submodels.

- `MMD_testC_Setup`/`MMD_testP_Setup`:

`SUBROUTINE MMD_testC_Setup`		`(nx, ny)`	
name	type	intent	description
mandatory arguments:			
nx	INTEGER	IN	number of grid points in first horizontal dimension
ny	INTEGER	IN	number of grid points in second horizontal dimension

`SUBROUTINE MMD_testP_Setup`		`(Id, nx, ny, cdim_order)`	
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
nx	INTEGER	IN	number of grid points in first horizontal dimension
ny	INTEGER	IN	number of grid points in second horizontal dimension
cdim_order	CHARACTER(LEN=4)	IN	*axis string* to be used for the test array

As a first step in both models the `test_array` is allocated. This is easy for the child model (subroutine `MMD_testC_Setup`) as the `test_array` is simply allocated according to the local horizontal dimensions (as provided by the child submodel) and the required additional parameter dimensions (N=8) of the array on each child PE. The setup of the parent model (in `MMD_testP_Setup`) is more demanding. First, the local dimensions are not necessarily equal on all parent PEs. Therefore the maximum local horizontal dimension length of all parent model PEs needs to be determined. Second, as the order of the coordinate axes can be different in different models, different axes orders (e.g., `'XZNY'` or `'XYNZ'`)[15] have to be distinguished. Thus, more than one `test_array` is required during the initialisation phase of the parent model:

  - First, a `global_array` allocated only on the PE with `m_model_rank=0` is used during the initial phase as the filling of the `test_array` only works for a global field (see subroutine `MMD_testP_Fill`).
  - Second, the global field is scattered to the individual tasks. Thus a local `test_array` dimensioned by the maximum horizontal dimensions is required during the initial phase.

[15]Note: at the moment only ECHAM and COSMO are implemented as parent models. For the coupling of other models other axis order strings can be taken into account in future.

– Finally the exchanged field, which is a local `test_array` dimensioned by the respective local dimensions, is filled by the intermediate local `test_array` dimensioned by the maximum (of all parent PEs) horizontal dimensions.

- `MMD_testP_GetTestPtr`/`MMD_testC_GetTestPtr`:

`SUBROUTINE MMD_testP_GetTestPtr (Id, p, axis, ldim)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
p	REAL(dp),DIMENSION(:,:,:,:)	POINTER	returned POINTER of `test_array`.
axis	CHARACTER(LEN=4)	OUT	*axis string* of test array
ldim	INTEGER, DIMENSION(4)	OUT	local dimensions of test array

`SUBROUTINE MMD_testC_GetTestPtr (p, axis, ldim)`			
name	type	intent	description
mandatory arguments:			
p	REAL(dp),DIMENSION(:,:,:,:)	POINTER	returned POINTER of `test_array`
axis	CHARACTER(LEN=4)	OUT	*axis string* of test array.
ldim	INTEGER, DIMENSION(4)	OUT	local dimensions of test array

Because the `test_array`s are handled as normal *exchange fields*, the POINTER to the `test_array`s and the dimension information about `test_array`s must be provided to the child and the parent submodel using the subroutines `MMD_testP_GetTestPtr` and `MMD_testC_GetTestPtr`, respectively.

- `MMD_testP_Fill`:

`SUBROUTINE MMD_testP_Fill (Id, index_list, lon, lat)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model
index_list	INTEGER, DIMENSION(6)	IN	sextuple describing the connection between one grid point on one parent PE with one grid point of a child PE; This list contains: index pair on parent PE, index pair on child PE, number of child PE, number of parent PE.
lon	REAL(dp)	IN	longitude of the grid point described by `index_list`
lat	REAL(dp)	IN	latitude of the grid point described by `index_list`

So far parent and child required similar preparations. The rest of the initialisation phase is purely work on the parent model side as the `test_array` needs to be filled with the `index_list` information. The subroutine `MMD_testP_Fill` fills the `global_array`. It is called separately for each sextuple in the `index_list`. As described above, the `index_list` sextuple is copied to the third to eighth entry of the `global_array`, whereas the longitude and the latitude of the global grid are assigned to the first two entries (see Table 3).

- `MMD_testP_FinishFill`:

`SUBROUTINE MMD_testP_FinishFill (Id)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of child model in the child list of this specific parent model

At the end of the filling procedure within the subroutine `MMD_testP_FinishFill` the `global_array` is scattered to the local fields on each PE and afterwards copied to the `test_array`, which is allocated to

the exact size of the local fields. The global and the first local test array are deallocated afterwards. Now the test field is ready for the data exchange. The field is automatically exchanged together with the other fields.

- `MMD_testC_Compare`:

`SUBROUTINE MMD_testC_Compare`		`(lon, lat, istat [, PrintUnit])`	
name	type	intent	description
mandatory arguments:			
lon	REAL(dp),DIMENSION(:,:)	IN	original *in-field* longitude
lat	REAL(dp),DIMENSION(:,:)	IN	original *in-field* latitude
istat	INTEGER	OUT	error/status flag
optional arguments:			
PrintUnit	INTEGER	IN	unit for diagnostic output

After the child model PEs received all buffers from all parent model PEs, the test routine `MMD_testC_Compare` is called. Here the geographical coordinates and the indices are compared as explained above. If an error occurs, the subroutine writes some error output to facilitate the error search and afterwards an error flag is returned triggering a termination of the model simulation.

- `MMD_testP_FreeMem` / `MMD_testC_FreeMem`:

`SUBROUTINE MMD_testP_FreeMem`		`(ChildId)`	
name	type	intent	description
mandatory arguments:			
ChildId	INTEGER	IN	index of child model in the MMD model setup

`SUBROUTINE MMD_testC_FreeMem`	`()`

At the end of a simulation the memory allocated by `mmd_test` is deallocated within the subroutines `MMD_testP_FreeMem` and `MMD_testC_FreeMem`.

3.2 The C part of the MMD library

Even if mixing different programming languages within the same library is not desirable, this way was chosen for MMD. The main library interface which is addressed by the parent and child submodels (MMD2WAY_CHILD and MMD2WAY_PARENT) should be easily expandable for a scientific user. Even more important: the data fields filled on the child and parent model side are accessed by Fortran95 POINTERs. These POINTERs can point to non-contiguous hyperslices of higher dimensioned TARGETs. In contrast, C POINTERs provide access to a field by pointing to the memory address where this field starts, assuming that the field is stored contiguously in the memory. The MMD2WAY sub-submodels provide Fortran95 POINTERs, which need not to be contiguous in memory (the most prominent example is a POINTER to one tracer). Therefore, the library part remapping the exchanged data to the fields needs to be written in Fortran95. Nevertheless, C-code can not completely be avoided within the library as the MPI function `MPI_alloc_mem` is indispensable for the library and this function is not usable in Fortran95. As most of MESSy is written in Fortran95 the C part is kept as short as possible. It only contains those functions required for the dimensioning and the allocation of the exchange buffers and the data exchange itself.

3.2.1 `cfortran.h`

As the MMD library combines Fortran95 and C, the header file `cfortran.h` is used fpr C to Fortran95 binding.

3.2.2 `mmdc_util.h`

This header file contains the definitions used in the C part of the MMD library. The structure `ModelDef` comprises the information required for data exchange between the current and the *remote model*:

```
struct ModelDef {
  MPI_Aint             TotalBufferSize;/* Size of buffer of each PE     */
  MPI_Comm             model_comm;     /* Communicator of this model    */
  MPI_Comm             inter_comm;     /* Inter model communicator      */
  MPI_Comm             intra_comm;     /* Intra model communicator      */
  int                  model_rank;     /* Rank of this model            */
  int                  model_npes;     /* Number of PEs of this model   */
  int                  inter_npes;     /* Number of PEs of remote model */
  MPI_Win              BufWin;         /* MPI RMA windows               */
  struct BufDef        *buf;
};
```

- `TotalBufferSize` is the buffer size each PE has to send/receive to/from all *remote PEs*.
- The communicators: `model_comm` and `inter_comm` are the MPI-communicators required for the communication between the processes of the current model and the communication to the *remote model PEs*, respectively. `intra_comm` is the communicator for simultaneous communication of all PEs of the current and the *remote model*.
- The rank of each PE within the group communicator of the current model is `model_rank`.
- `model_npes` and `inter_npes` are the number of PEs used by the current and the *remote model*, respectively.
- `BufWin` is of MPI type `MPI_Win` and is required to control the access to the memory buffer used by the child and the parent model.
- The structure `BufDef` contains all information specific for one *remote model PE*:

  ```
  struct BufDef {
    int                  BufLen;       /* Length of buffer                  */
    int                  pBufLen;      /* Length of parent buffer           */
    long                 BufIndex;     /* Index in Send Buffer              */
    double               *SendBuf;     /* Pointer of Data in Send buffer    */

    struct BufDef      *next;
  };
  ```

  `BufDef` is a concatenated list with as many members as the number of *remote PEs*:

  - `BufLen` gives the length of the buffer exchanged with the respective *remote PE* requested by the child model,
  - `pBufLen` gives the length of the buffer exchanged with the respective *remote PE* requested by the parent model,
  - `BufIndex` is the index in the overall received/sent buffer from which on the respective buffer starts.
  - `SendBuf` is the POINTER to the sent/received buffer.
  - `next` points to the next list element or is nullified for the last element.

  Sect. 3.3 illustrates the meaning of these variables.

3.2.3 mmdc_util.c

```
double MMDc_U_Time                          ()
```

The file `mmdc_util.c` contains one function only. `MMDc_U_Time` inquires the system clock and hands back a number of type double representing the current system time.

3.2.4 mmdc_parent.c

mmdc_parent.c contains seven functions:

- MMDc_P_Init:

void MMDc_P_Init		(model_comm, inter_comm, *npes)	
name	type	intent	description
arguments:			
ChildId	int	IN	index of child model in the MMD model setup
model_comm	int	IN	internal model communicator
inter_comm	int	IN	model communicator to *remote model*
npes	int	OUT	number of *remote PEs*

  First, the communicators within the C environment are set up during the initialisation phase by the function MMDc_P_Init. Additionally, this function provides the number of *remote PEs* to the Fortran95 part of the library.

- MMDc_P_SetInd_and_Mem:

void MMDc_P_SetInd_and_Mem		(ChildId, *bufsize)	
name	type	intent	description
arguments:			
ChildId	int	IN	index of child model in the MMD model setup
bufsize	int, DIMENSION(:)	IN	length of buffer exchanged with each *remote PE*

  The parent side of the MMD library has the important task to calculate the association of the PEs and grid points of the current and the *remote model*. The index_list is prepared within the parent submodel MMD2WAY_PARENT. The Fortran95 part of the MMD library calculates the dependencies between the parallel decomposed parent and child model grids. From this, the length of the buffer each parent PE exchanges with each child PE is calculated. This is the only information required by the C part of the library.

  - The function MMDc_P_SetInd_and_Mem stores this information within the structure component &Childs[*ChildId-1]->buf->BufLen.
  - The total buffer length (&Childs[*ChildId-1]->TotalBufferSize) is calculated by summing up the individual buffer lengths exchanged with the *remote PEs*.
  - The buffers exchanged with each child PE are aligned to one large one-dimensional buffer. The information, at which index in this large buffer the array for the individual child PEs starts, is stored in the structure component &Childs[*ChildId-1]->buf->BufIndex.
  - Each of these indices is sent to the respective child PE, to enable the child PE to locate the corresponding buffer part.
  - The total buffer length (&Childs[*ChildId-1]->TotalBufferSize) is used to allocate the exchange buffer in the correct size by the MPI function MPI_alloc_mem.
  - For the provision of the data, a window is created with MPI_Win_create and its handle is stored in &Childs[*ChildId-1]->BufWin.
  - Afterwards the POINTERs to the starting addresses of each buffer exchanged with every child PE (&Childs[*ChildId-1]->buf->SendBuf) is stored for later use.

- `MMDc_P_SetInd_and_Mem_2way`:

`void MMDc_P_SetInd_and_Mem`		`(ChildId, *bufsize, *parbufsize)`	
name	type	intent	description
arguments:			
ChildId	int	IN	index of child model in the MMD model setup
bufsize	int, DIMENSION(:)	IN	length of buffer exchanged with each *remote PE* requested by the child model
parbufsize	int, DIMENSION(:)	IN	length of buffer exchanged with each *remote PE* requested by the parent model

The subroutine `MMDc_P_SetInd_and_Mem_2way` works in the same way as `MMDc_P_SetInd_and_Mem`, only that the size of the allocated buffer is determined by the maximum of the buffer sizes required for the child and the parent model.

- `MMDc_P_FillBuffer`:

`void MMDc_P_FillBuffer`		`(ChildId, *PeId, *array)`	
name	type	intent	description
arguments:			
ChildId	int	IN	index of child model in the MMD model setup
PeId	int	IN	Id of the *remote PE* the actual buffer is sent to
array	double	IN	data to be copied to the buffer memory space

Within the time loop in the function `MMDc_P_FillBuffer`, the data provided as parameter `array` is copied to the correct location of the buffer (i.e., to the memory space, which is accessible from parent and child model).

- `MMDc_P_SetBarrier`:

`void MMDc_P_SetBarrier`		`(ChildId)`	
name	type	intent	description
arguments:			
ChildId	int	IN	index of child model in the MMD model setup

When the buffer is completely filled, i.e., each parent PE filled the buffers for all child PEs, a barrier is set by the function `MMDc_P_SetBarrier` to prevent the parent from overwriting or reading the buffer before it was copied by the child model. After the child model read the data and, in case of 2-way coupling, filled the buffer again, the barrier is released by the child model and the parent can read / fill the buffer again.

- `mmdc_P_GetBuffer`

`void MMDc_P_GetBuffer`		`(*ChildId, *PeId, *array)`	
name	type	intent	description
arguments:			
ChildId	int	IN	index of respective child model
PeId	int	IN	number of *remote PE*
array	double	OUT	array of data copied from the common memory space

The function `MMDc_P_GetBuffer` is the one actually exchanging the buffer between child and parent model, i.e., the data in the buffer is copied to a 1-dimensional array, which in a second stage in the Fortran95 part of the library, is copied to the *in-fields* processed by the parent submodel MMD2WAY_PARENT.

- `MMDc_P_GetWaitTime`:

`void MMDc_P_GetWaitTime`		`(ChildId,*WaitTime )`	
name	type	intent	description
arguments:			
ChildId	int	IN	index of child model in the MMD model setup
WaitTime	double	OUT	time span the parent model had to wait

`mmdc_parent.c` comprises the additional function `MMDc_P_GetWaitTime`. It measures the time span between the call of the `MMDc_P_FillBuffer` function and the release of the buffer by the child model.

3.2.5 `mmdc_child.c`

The work the child model has to perform is split into three parts:

1. The initialisation of the MMD library child model side.

`void MMDc_C_Init`		`(*model_comm, *inter_comm, *npes)`	
name	type	intent	description
arguments:			
model_comm	int	IN	internal model communicator
inter_comm	int	IN	model communicator to child model
npes	int	OUT	number of *remote PEs*

   Function `MMDc_C_Init` is called from the Fortran95 subroutine `MMD_C_Init` and initialises the MMD child C environment, i.e., the C model communicators (`model_comm` and `inter_comm`) in agreement with the Fortran95 communicators. From these two, the intra communicator (`intra_comm`), the model rank `model_rank` and the number of the PEs in the current and the parent model (`model_npes` and `inter_npes`) are determined. Based on the number of parent PEs, the variable `buf` of type `BufDef` is allocated. The number of parent model PEs is output parameter of this function, to be used in the Fortran95 part of the library.

2. The calculation of the matrix relating the memory filled by each parent PE to the respective child PE. Different routines exist for 1-way and for 2-way coupling. As for 2-way coupling the memory exchanged in both directions needs to be taken into account.

   - `MMDc_C_SetInd_and_Mem`

`void MMDc_C_SetInd_and_Mem`		`(*bufsize)`	
name	type	intent	description
arguments:			
bufsize	int, DIMENSION(:)	IN	length of buffer exchanged per *remote PE* for data requested by the child model

     In function `MMDc_C_SetInd_and_Mem` the information about the length and the location of the buffer sections accessed by each specific child model PE within the exchanged buffers is stored.
     - The calling Fortran95 subroutine provides the buffer length, which is required as input and differs for each of the parent PEs for the current child model PE. This length is stored in the structure component `me->buf->BufLen`.
     - The parent model PE sends an index, which indicates, where the data for the respective child PE starts in the 1-dimensional array of data sent by the parent PE. This index is stored in the structure component `me->buf->BufIndex`.
     - Additionally, the maximum length of all buffer lengths exchanged by the current child PE (`maxbufsize`) is determined. This is the required buffer size on the child model side, because the child exchanges data only with one parent PE at once.
     - Thus, a buffer of respective size is allocate with `MPI_alloc_mem`.
     - For accessing the data provided by the parent PE, a window is created with `MPI_Win_create` and its handle is stored in `me->BufWin`.
     - Finally, the POINTER used to access the data of each parent PE is stored in `me->buf->SendBuf`.

- `MMDc_C_SetInd_and_Mem_2way`

`void MMDc_C_SetInd_and_Mem`			`(*bufsize, *parbufsize)`
name	type	intent	description
arguments:			
bufsize	int, DIMENSION(:)	IN	length of buffer exchange per *remote PE* for data requested by the child model
bufsize	int, DIMENSION(:)	IN	length of buffer exchange per *remote PE* for data requested by the parent model

Subroutine `MMDc_C_SetInd_and_Mem_2way` works in the same way as subroutine `MMDc_C_SetInd_and_Mem`. Only for the length of the memory buffer the maximum of both exchange buffers has to be considered.

3. The actual buffer exchange:

- `MMDc_C_GetBuffer`

`void MMDc_C_GetBuffer`			`(*PeId,*array)`
name	type	intent	description
arguments:			
PeId	int	IN	number of *remote PE*
array	double	OUT	array of data copied from the common memory space

The function `MMDc_C_GetBuffer` is the one actually exchanging the buffers, i.e., the data in the buffer is copied to a 1-dimensional array, which, in a second stage in the Fortran95 part of the library, is copied to the *in-fields* processed by the Child submodel.

- `MMDc_C_FillBuffer`

`void MMDc_C_GetBuffer`			`(*PeId,*array)`
name	type	intent	description
arguments:			
PeId	int	IN	number of *remote PE*
array	double	OUT	array of data copied to the common memory space

The function `MMDc_C_FillBuffer` is the one actually exchanging the data requested by the parent model. Therefore, the data provided as a 1-dimensional array by the Fortran95 interface is copied to the buffer, to be read by the parent model subroutine `MMDc_P_GetBuffer`.

- `MMDc_C_SetBarrier`:

`void MMDc_C_SetBarrier`	`(C)`

When the buffer is completely filled, i.e., each child PE filled the buffers for all parent PEs, a barrier is set within the function `MMDc_C_SetBarrier` to prevent the child from overwriting / reading the buffer before it is copied / newly filled by the parent model. After the parent, in case of 2-way coupling, read the data and filled the buffer again, the barrier is released by the parent and the child can read and fill the buffer again.

- `MMDc_C_GetWaitTime`:

`void MMDc_C_GetWaitTime`		`(*WaitTime)`	
name	type	intent	description
arguments:			
WaitTime	doubl	OUT	time span the parent model has to wait

Additionally, the function `MMDc_C_GetWaitTime` is provided, measuring the time from calling the subroutine `MMD_C_GetBuffer` until the parent sets the barrier indicating that the buffer is full. This is useful for benchmarking and run-time optimization.

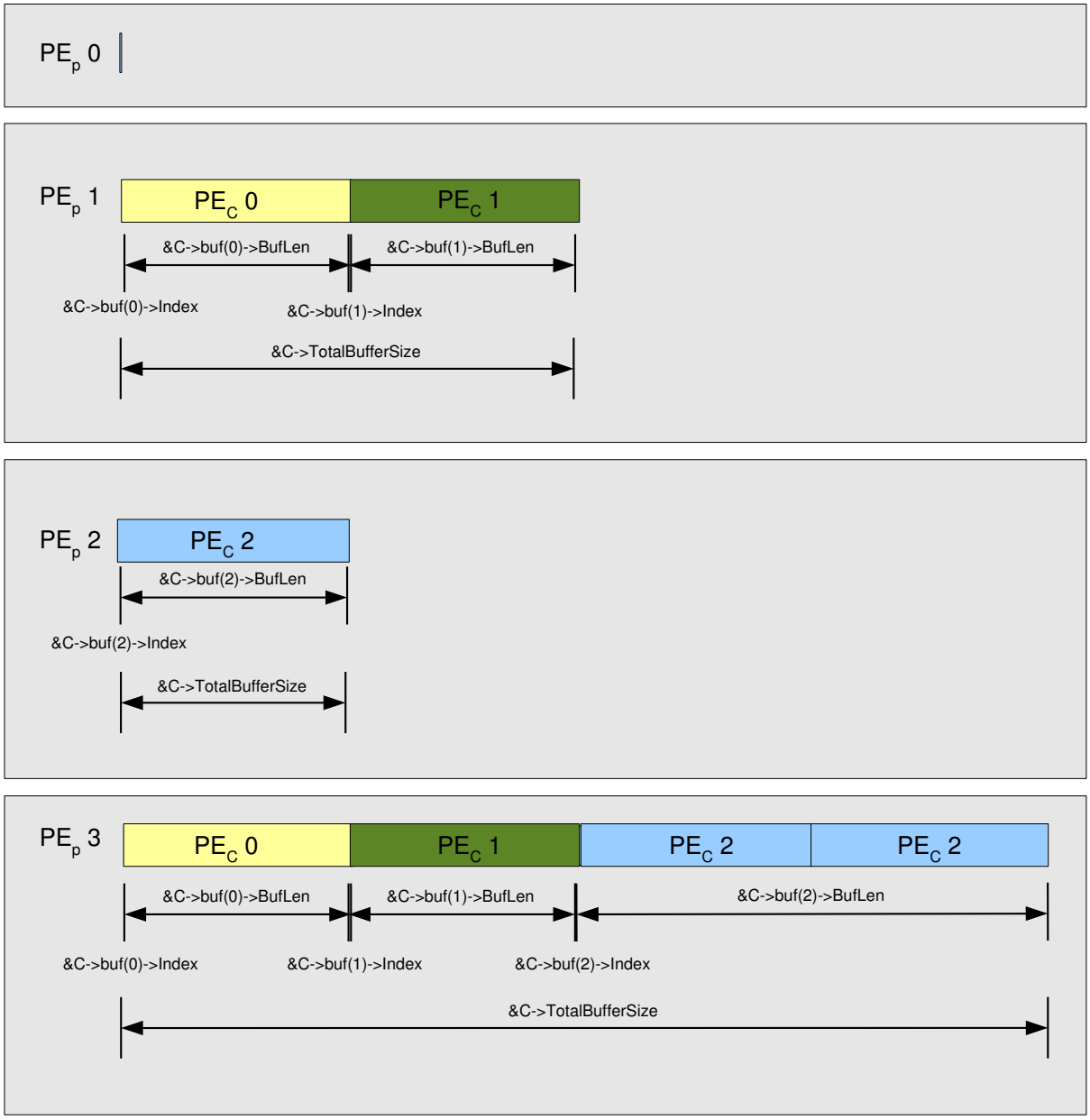


Figure 6: Example of a possible MMD exchange buffer layout.

3.3 Example

In this section the meaning of the index and length variables used in the structures are clarified with the help of the example shown in Fig. 3 and Table 1. The example is for the exchange from data from the parent to one child model only. Anyhow, the exchange in the other direction works exactly in the same way.

Figure 6 illustrates the buffer definition on the parent model side in the C part of the library. Each grey box depicts one parent PE. The first grey box is empty, as the child domain does not overlap with PE_p 0. Parent PE_p 1 contributes one grid point to child PE_c 0 and one to PE_c 1. This is illustrated by the coloured bars: yellow for child PE_c 0 and green for PE_c 1. Each of this buffer parts is `&Child[*ChildId-1]->buf->BufLen` long. In the figure the first part of the structure `&Child[*ChildId-1]` is abreviated by `&C`. As explained above `buf` is a concatenated list. The different elements of this list are depicted by indices in the figure, e.g. ,

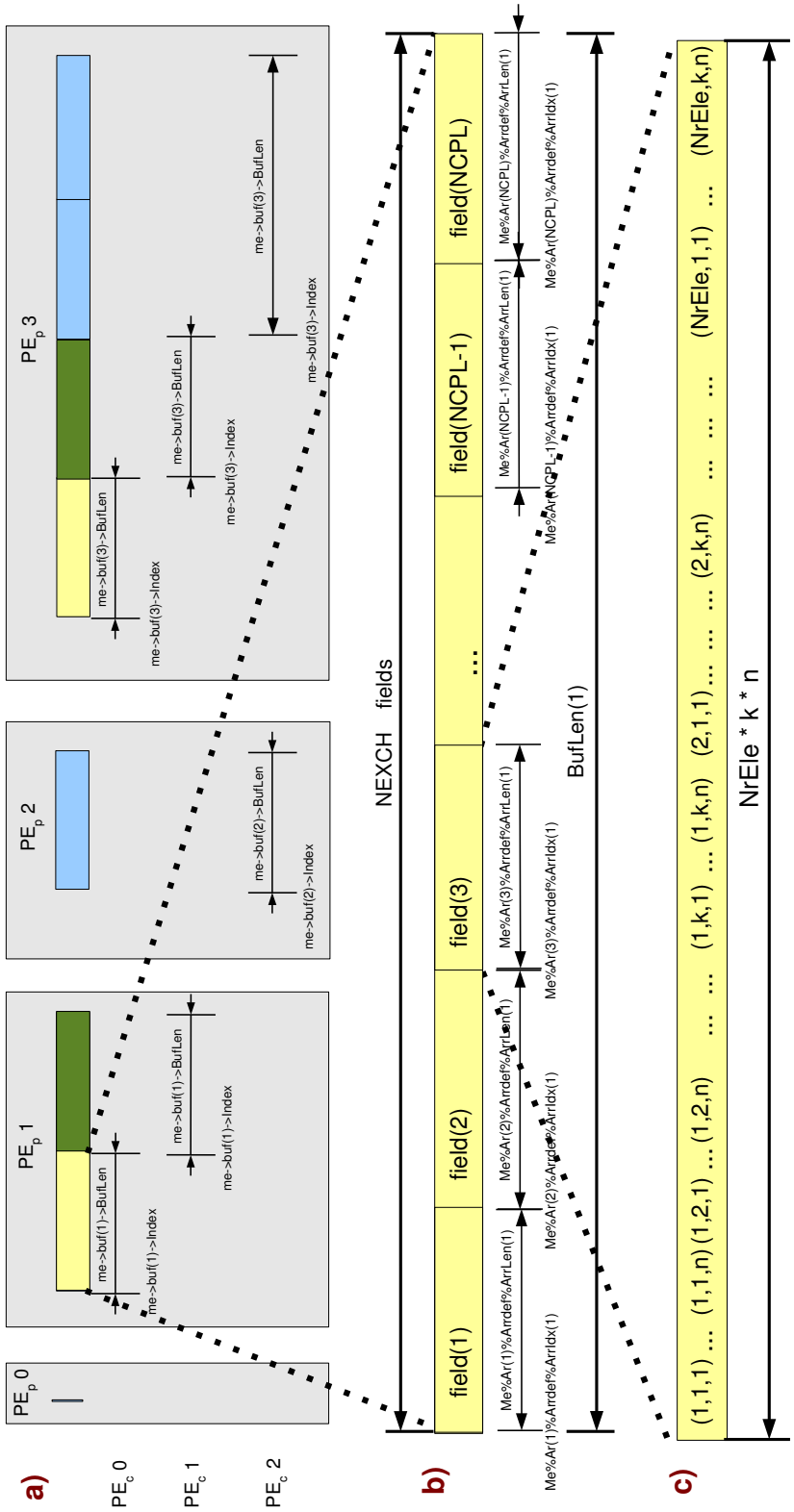


Figure 7: Example of a possible MMD exchange buffer layout.

`&C->buf(1)->BufLen` means the length of the buffer that is sent to PE_c 1. For the correct access to the data, it must be known, where the buffers for each of the child PEs starts. This is indicated by the respective indices `&C->buf( )->Index`. The overall length of the buffer sent by PE_p 1 is given by `&C->TotalBufferSize`.

Parent PE_p 2 only sends data to child PE_c 2. Thus `&C->TotalBufferSize` for PE_p 2 is equal to the length of the buffer sent to PE_c 2 (`&C->buf(2)->BufLen`). The buffer lengths attributed to the other child PEs are set to zero.

Parent PE_p 3 sends data to all three child PEs: one grid point each to child PE_c 0 and PE_c 1 and two grid points to PE_c 2. Thus `&C->buf(2)->BufLen` is twice as large as `&C->buf(0)->BufLen` or `&C->buf(1)->BufLen`.

Figure 7a) shows the same for the child model side. This figure part is constructed like a table. The parent PEs build the columns and the Child PEs the rows. The grey boxes indicate again the parent PEs. As PE_p 0 does not send any data, all variables are zero on all child PEs. Child PE_c 0 receives data from PE_p 1 and PE_p 3. The buffer sent by PE_p 1 is `me->buf(1)->BufLen` long and starts at position `me->buf(1)->BufIndex`. Note that the index 1 of `buf` referes to PE_p 1. Correspondingly, the buffer received from PE_p 3 starts at `me->buf(3)->BufIndex` and its length is `me->buf(3)->BufLen`. The `maxbufsize` for PE_c 0 is equal to `me->buf(1)->BufLen` and `me->buf(3)->BufLen`, as they are equally long.

The respective variables for child PE_c 1 are defined accordingly. Child PE_c 2 receives data from the two parent processes PE_p 2 and PE_p 3. The definitions are similar, different is `maxbufsize`, which is obviously set to `me->buf(3)->BufLen` as PE_c 2 receives two grid points from PE_p 3 but only one from PE_p 2.

The C part needs to deal only with the buffers exchanged between one parent and one child PE. While this part is illustrated in Fig. 7a), parts 7b) and 7c) of the figure deal with the variable definitions in the Fortran95 part of the library, where the single exchanged fields (Fig. 7b) and individual horizontal grid elements of each field (Fig. 7c) are addressed.

Figure 7b) illustrates the order of the single fields within one buffer dealt with by C. Per exchanged buffer (i.e., per child and per parent PE) the fields are stored one after the other in the order of their definition. The number of fields is always the full number of *exchange fields* (`NEXCH`). The length of the individual fields can vary, as the vertical and the number dimension can differ[16]. The individual length of each field is saved in the variable `Child(Id)%Ar(ix)%Arrdef%ArrLen(ip)`. Where `ix` is the number of the field. This is again a pseudo-index used for illustration, as `Child(Id)%Ar` is a concatenated list. The index `ip` indicates the respective parent PE. `Me%Ar(ix)%Arrdef%ArrLen(ip)` can be different for each parent PE, because the number of elements exchanged with each parent PE (`Me%PEs(ip)%NrEle`) can be different for each parent PE. As the array lengths can vary, the start index of each of the fields within the exchanged buffer is saved in the variable `Me%Ar(ix)%Arrdef%ArrIdx`. `BufLen(ip)` in `mmd_child.f90` is defined as the sum over `ix` in `Me%Ar(ix)%Arrdef%ArrLen(ip)`. `BufLen(ip)` equals `me->buf(ip)->BufLen` in `mmdc_client.c`.

Figure 7c) shows the sequence of the single data points of one field as aligned by the packing algorithm. The fastest varying index is the number dimension (`n`), the second fastest is the vertical dimension (`k`). The horizontal dimension (from 1 to `NrEle`) varies most slowly.

A Glossary

- *attributes*: *Attributes* represent time independent, scalar characteristics, e.g., the measuring unit.
- *axis string*: It is a CHARACTER of length 4, it is defined for each *representation*, indicating which rank is associated to which dimension. For instance, 'XY–' indicates a horizontal 2D field in grid point space.
- *channel*: The generic submodel CHANNEL manages the memory and meta-data and provides a data transfer and export interface. A *channel* represents sets of "related" *channel objects* with additional meta information. The "relation" can be, for instance, the simple fact that the *channel objects* are defined by the same submodel.
- *channel object*: It represents a data field including its meta information and its underlying geometric structure (*representation*), e.g., the 3-dimensional vorticity in spectral *representation*, the ozone mixing ratio in Eulerian *representation*, the pressure altitude of trajectories in Lagrangian *representation*.

[16]Note: the horizontal dimensions must be the same for all fields exchanged between one Parent and one Child PE, i.e., `Me%PEs(ip)%NrEle`.

- *dimensions*: They represent the basic geometry of one dimension, e.g., the number of latitude points, the number of trajectories, etc.
- *exchange field*: An *exchange field* is a field requested within the `&CPL_CHILD_ECHAM` or `&CPL_CHILD_COSMO` namelist in the `mmd2way.nml` namelist file and provided by the parent to the child. An *exchange field* can either be a field which is remapped and copied to a child variable, or a field required for the grid mapping itself.

  For the 2-way coupling the term is also used for the fields the parent receives from the child model.
- *in-field*: The *in-fields* are those fields provided by the parent or *driving model*, which are still defined on the parent grid, but on the child side. In other words, *in-fields* are the *exchanged fields* before the grid mapping.

  For the 2-way coupling this term is also used for the fields received by the parent model. In this case the field have already been mapped to the parent grid, but these fields are only defined on the exchanged area.
- *in-grid*: grid on which the *in-fields* are defined.
- *master parent*: The *master parent* or *patriarch* is the coarsest model in a model cascade, i.e., that model that has no parent model itself. In the MMD library namelist this model is indicated by a "-1" as associated parent model. In most cases this is a global model. The *patriarch* determines the timing of the entire model cascade.
- *out-grid*: The *out-grid* is a subpart of the parent model grid, defined by the child submodel MMD2WAY_CHILD. This is the target grid for the remapping of the child model fields to the parent grid before the remapped data is sent back to the parent.
- *patriarch*: The *patriarch* or *master parent* is the coarsest model in a model cascade, i.e., that model that has no parent model itself. In the MMD library namelist this model is indicated by a "-1" as associated parent model. In most cases this is a global model. The *patriarch* determines the timing of the entire model cascade.
- *Receiver*: short for *receiving model*
- *receiving model*: the model receiving the data. In case of 1-way coupling the child model (client) is always the *receiving model*, while the parent model (server) is always the *sending model*.
- *remote model*: the "other" model in a communicating child-parent model pair; i.e., for the child the parent, for the parent the respective child
- *remote PE*: the "other" PE in a pair of child and parent model PEs exchanging data. For example, parent PE_p 2 is sending data to child PE_c 4: in this case PE_c 4 is the *remote PE* for parent PE_p 2 and vice versa.
- *representation*: It describes multidimensional geometric structures (based on *dimensions*), e.g., Eulerian (or grid point), spectral, Lagrangian.
- *Sender*: short for *sending model*
- *sending model*: the model sending the data. In case of 1-way coupling the child (client) model is always the *receiving model*, while the parent (server) model is always the *sending model*.

References

Jöckel, P., Kerkweg, A., Pozzer, A., Sander, R., Tost, H., Riede, H., Baumgaertner, A., Gromov, S., and Kern, B.: Development Cycle 2 of the Modular Earth Submodel System (MESSy2), Geosci. Model Dev., 3, 717 – 752, 2010.