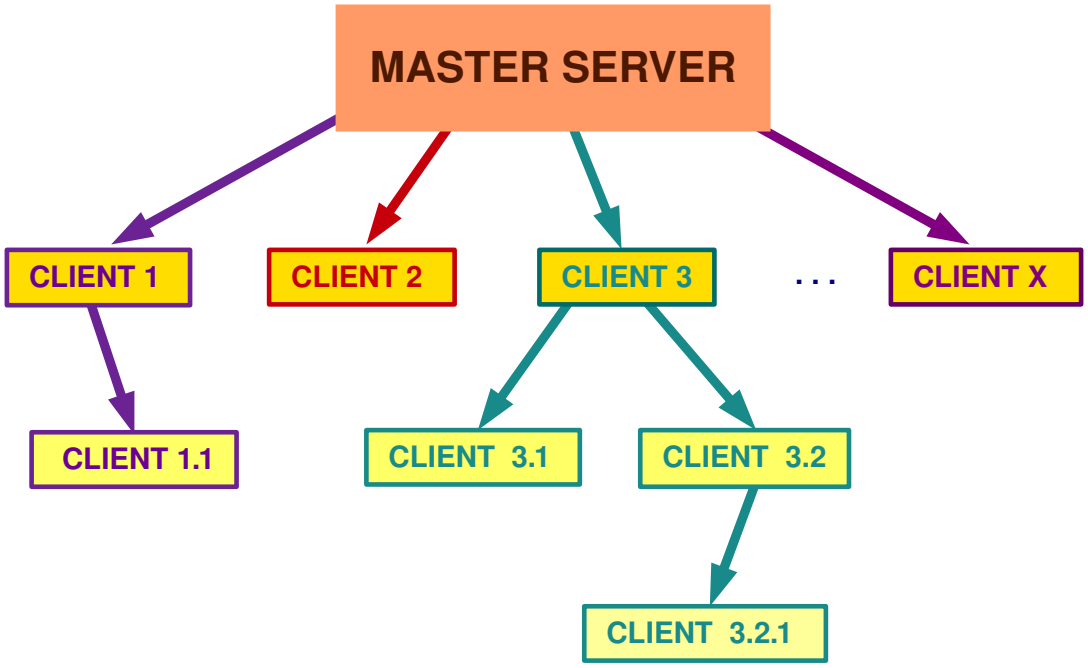

Multi-Model-Driver (MMD) Library Manual

Version 1.1

Astrid Kerkweg[1] & Patrick Jöckel[2]

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

This manual is available as electronic supplement of our article "The 1-way on-line coupled atmospheric chemistry model system MECO(n): Part II: On-line Coupling " in Geosci. Model Dev. (2011), available at: http://www.geosci-model-dev.net

Date: November 2, 2011

Contents

1 Introduction

The Multi-Model-Driver (MMD) was developed to carry out the on-line coupling between models, i.e., two or more models run concurrently within the same message passing interface (MPI) environment and exchange data by MMD. Following a client-server approach, the server (driving model) provides data to a client model. For the data exchange the sending and the receiving side must be distinguished. In the following the model sending the information is called "server" and the model receiving data is named "client". The other model in the corresponding model pair is further denoted as *remote model*[1], i.e., the *remote model* of the server is the client model and vice versa.

MMD consists of three parts:

- the Multi-Model-Driver (MMD) library provides all routines necessary for the data exchange between executables of basemodels,
- the server submodel (MMDSERV) providing the data to the client model(s) and
- the client submodel MMDCLNT receiving the data from the server and processing it to provide the data to the client model in an appropriate way.

While the MMD library manages the communication between the server and the client submodels, the client submodel MMDCLNT controls the data exchange via namelist.

In this manual we focus on the technical structure and functionality of the MMD library. A description of MMDCLNT and MMDSERV 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 communication.

The MMD library is build in a way that an arbitrary number of models can be run concurrently within the same MPI environment. Each model can be server for an arbitrary number (including zero) of other models and is client of exactly one model. The only exception is the global or "coarsest" model of the setup. This one drives all the other models (direct or indirect via other clients), but is not client of any other model itself. We call this model *master server*. The figure on the front page of the manual gives an example for a possible model cascade. Here, the *master server* "serves" 4 clients (CLIENT 1,2,3 and X). CLIENT 1 and 3 are server for other clients as well. CLIENT 1 has one client (1.1) and CLIENT 3 serves two clients (3.1 and 3.2). CLIENT 3.2 is again server model for CLIENT 3.2.1. The clients of one server are completely independent of each other.

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

The MMD library is mainly written in Fortran95 and partly in C. On the one hand this is necessary, 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 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 server and client specific routines organised in modules USEd by the MMDSERV and MMDCLNT submodels, respectively, and it contains partly routines USEd by the server and client basemodels directly. The names of the modules and the subroutines or functions indicate, whether the language is C or Fortran95. C files or functions start with `'mmdc_'`, while Fortran95 files or routines begin with `'mmd_'`. Fig. 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_client.f90`, `mmd_server.f90` and `mmd_test.f90` are directly used by the client and server submodels (MMDCLNT and MMDSERV, respectively). All three module files in turn USE the three modules `mmd_mpi_wrapper.f90`, `mmd_handle_communicator.f90` and `mmd_utilities.f90`.

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

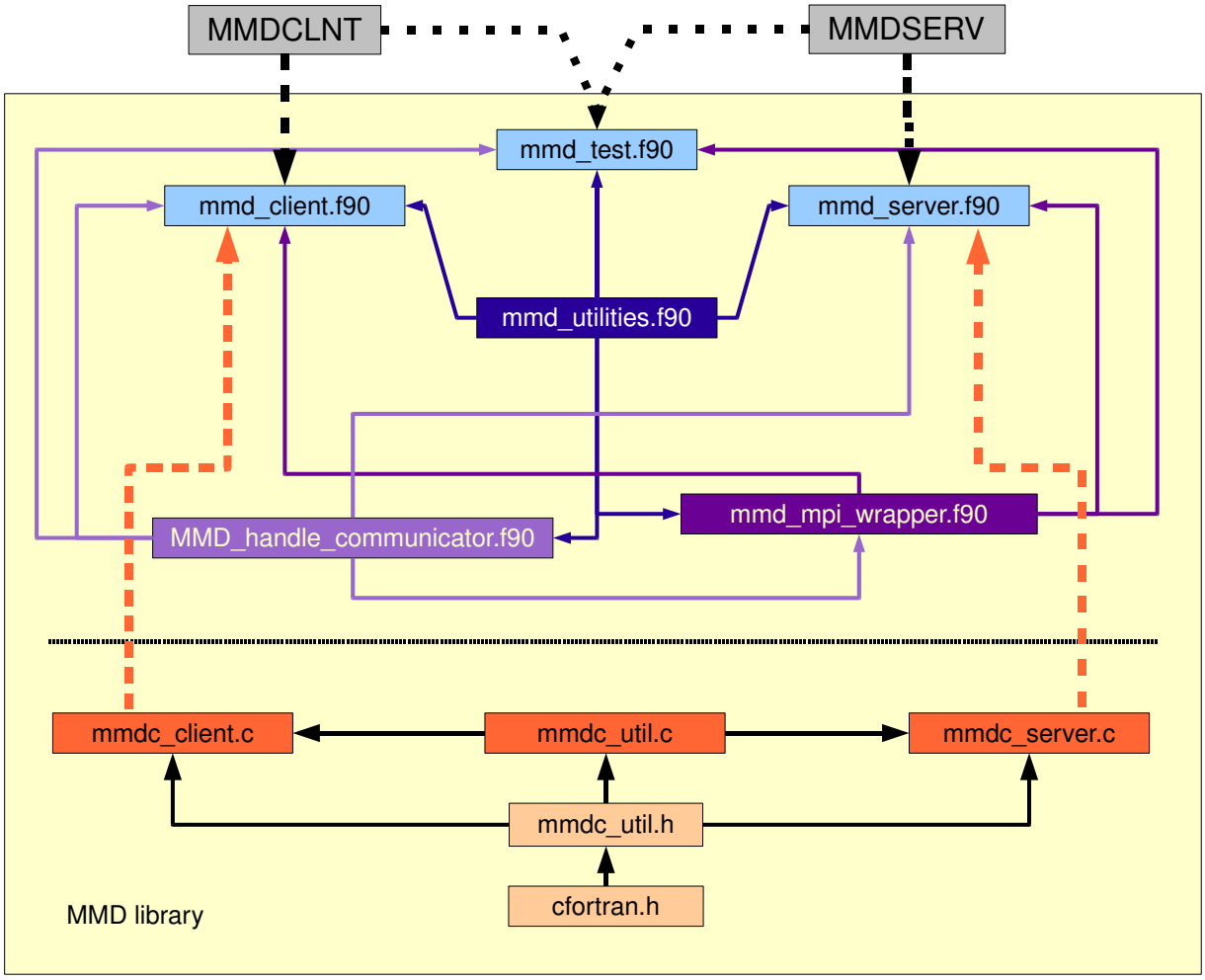

Figure 1: Hierachy of the MMD module files. Bluish colours indicate modules written in Fortran95, reddish colours denote C files. Arrows point in the direction of USagE, e.g., `mmd_utilities.f90` is used by all other Fortran95 modules.

- `mmd_mpi_wrapper.f90` supplies high level interface routines, which simplify the communication between server and client 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.
- Finally, `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_client.c` and `mmdc_server.c` contain the client and server 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 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 work flow 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 work flow in the MMD library

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

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[2] (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 models 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 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 within the MESSy submodels carrying out the coupling (MMDCLNT and MMDSERV).

- In the server specific subroutine `MMD_S_Allocate_Client` the variable `Clients` of a TYPE containing all required information about the client models and about the requested data is allocated to the actual number of client models of each individual server. For instance, in the example on the front page the *master server* requires space for 4 clients, whereas CLIENT 1 provides data to only one client.
- In the subroutines `MMD_S_Init` and `MMD_C_Init` the MMD internal variables containing the information about the *remote model(s)* in form of structures are allocated in the correct size according to the coupling layout. Furthermore, the setup routines for the C-core of MMD, `MMDc_C_Init` and `MMDc_S_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 client model provides the list of server and client *channel* and *channel object* names of the required data arrays and the corresponding *representation* to the MMD library by calling the subroutine `MMD_Set_DataArray_Name`. Within the library the client *channel* and *channel object* names are stored within variable `Me`, which is of TYPE `ClientDef`,

[2]Here equivalent to MPI tasks.

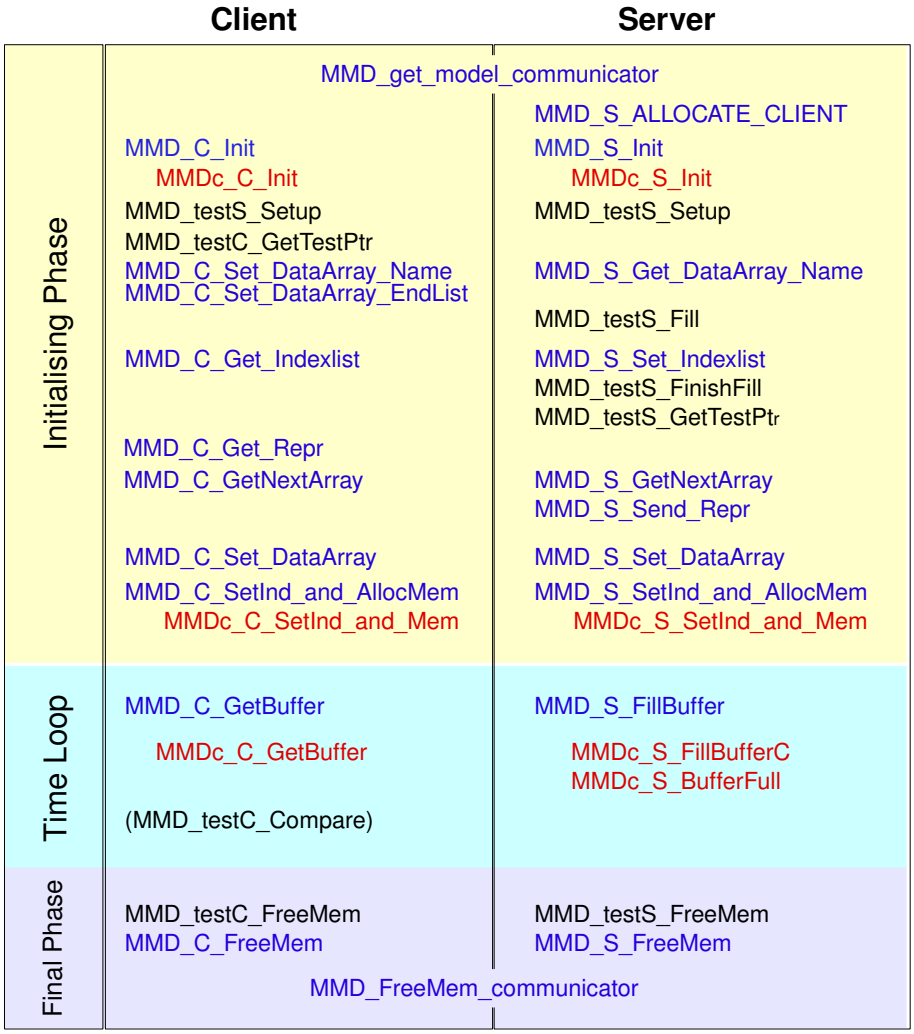

Figure 2: Call tree of the MMD library: blue are the names of the client and server specific Fortran95 routines of the MMD library, red are the C functions, whereas the black colour indicates the routines required for the test setup (written in Fortran95).

i.e., the overall coupling information structure for the client part of the library (see Sect. 3.1.1). Furthermore, the server *channel* and *channel object* names and their *representation*s are sent to the server model. On the server side the data is received and processed within the subroutine `MMD_S_Get_DataArray_Name`. Here, the *channel* and *channel object* names of the server *exchange fields* and their *representations* are stored in the information structure (`Clients`) for the respective client. These information will be accessed later on, with the use of the functions `MMD_S_GetNextArray` and `MMD_C_GetNextArray` on server and client side, respectively.

- Additional preparations are performed before the data exchange can take place:
  - First, the information is required, which server PE exchanges which parts of its horizontal grid with which client PE and their location in the client grid. Hereafter, this information is refered to as "`index_list`". It is provided by the server submodel MMDSERV to the MMD li-

brary as parameter to the subroutine `MMD_S_Set_Indexlist`. Within this subroutine the list is evaluated and sent to the client model. The client model receives and processes this list within the subroutine `MMD_C_Get_Indexlist`. For further details see the description of the subroutine `MMD_S_Set_Indexlist` in Sect. 3.1.5.

- Second, a test is initiated for checking the data exchange. For this the subroutines with the names starting with `'MMD_test'` are used.
- Third, the library needs to connect *channel* and *channel object* to the (Fortran95-)POINTERs.
  - On the server side, the list of the names of the *exchange fields* established by the subroutine `MMD_S_Get_DataArray_Name` is provided to MMDSERV by the subroutine `MMD_S_GetNextArray`. For each of the required fields the MESSy submodel MMDSERV provides a 4D-POINTER to the memory of the respective *exchange field* as parameter to the subroutine `MMD_S_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 PE to all client PEs. This number is used to actually allocate the required memory via MPI by the C function `MMDc_S_SetInd_and_Mem`, which is called via the Fortran95 subroutine `MMD_S_SetInd_and_AllocMem`.
  - For the client 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 similar. The *dimension* and *axis string* information is used during the integration phase to re-order the 1D-array received from the server into the 4D data arrays as requested by the client 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 (server) and read (client) within the time loop. First the server has to fill the buffer. This is done by the MMD library subroutines `MMD_S_FillBuffer` and `MMDc_S_FillBuffer` (see Sect. 3.1.5). After the buffer is filled, it is made accessible for the client side and a barrier is set to the server side by the subroutine `MMD_S_BufferFull`. Consequently, the buffer can now be read by the client part of the MMD library. 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*. After the buffers are completely read, the barrier set from the server is released by the client and a new barrier is set to prevent the client from accessing the same buffer twice.

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 client. 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 within the MMD library data structures is released in `MMD_C_FreeMem` and `MMD_S_FreeMem` for client and server 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 precise between the coupling submodels MMDCLNT and MMDSERV) 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_client.f90`, `mmd_server.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_ULONG   = 256
! ****************************************************************

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

INTEGER,PARAMETER,PUBLIC :: MMD_MAX_MODELL   = 64
INTEGER,PARAMETER,PUBLIC :: MMD_MPI_REAL = MPI_DOUBLE_PRECISION
! ****************************************************************
```

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 client MESSy submodel for the coupling (MMDCLNT) the server type has to be known, as some parts of the interpolation/coupling procedure depend on the server type. The INTEGER PARAMETERs `MMD_ServerIsECHAM` and `MMD_ServerIsCOSMO` are

used to indicate, whether the server 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_MODELL` determines the number of models maximally handled by MMD. If more than 64 models shall be run in parallel by MMD, this number needs to be increased within the code.

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

```
TYPE ClientDef
   ! NUMBER OF CLIENT PEs
   INTEGER                              :: inter_npes = 0
   ! CLIENT Id
   INTEGER                              :: ClientId   = 0
   ! STRUCTURE FOR EACH PE
   TYPE(PeDef), DIMENSION(:), POINTER :: PEs
   ! INDEX LIST OF SERVER 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 ClientDef
```

This structure defines the setup of exactly one client. Therefore the client side needs one scalar variable of the type of these structures, whereas each server needs an array dimensioned by the number of clients. `inter_npes` gives the number of processes (PEs) used by the *remote model*, i.e., on the client side the number of server PEs is stored, whereas on the server side the number of client PEs is of interest.

`ClientId` is the identification number (ID) of the client 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 `ClientId=8`. The `ClientId` is only used on the server side.

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 PE*[3]: For a client `NrEle` is the number of grid points the client PE receives from a specific server PE. For a server `NrEle` is the number of grid points, which this server PE provides to a specific client 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 client, `locInd` contains the index pairs of the local decomposed raw field as received from the server (here after denoted *in-field*), whereas for the server `locInd` stores the indices in its own local decomposed model domain.

Fig. 3 together with Table 1 illustrates these dependencies:

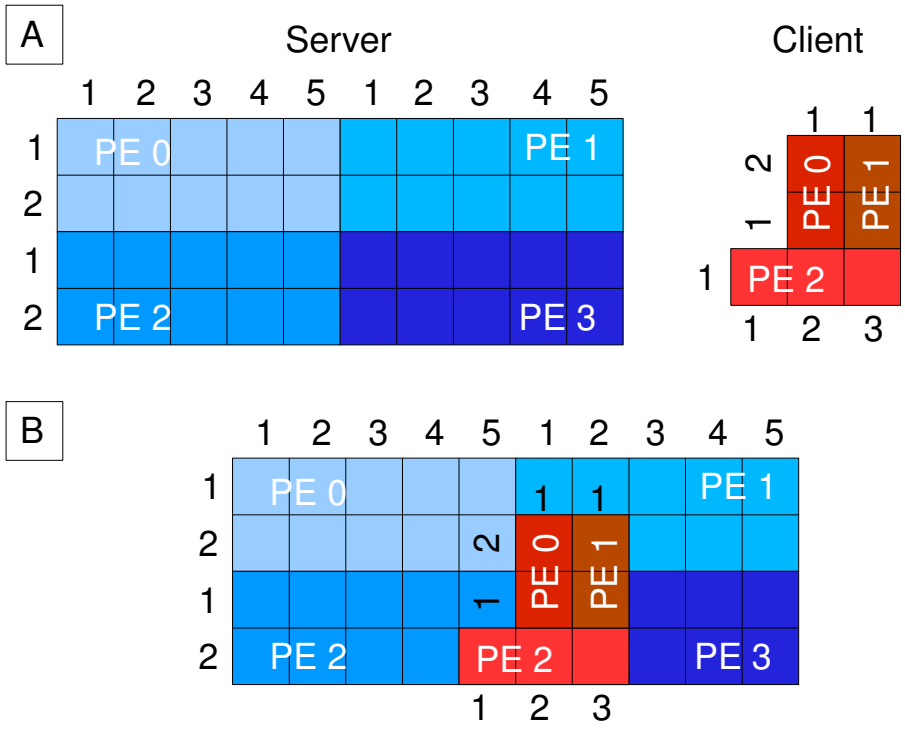

Figure 3: Illustration of possible model domain overlaps and the definition of the variable `NrEle`. A detailed explanation is provided in the text.

server PE	client PE	NrEle	(x,y) index pairs server	(x,y,) index pair client
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 server and client PEs. `NrEle` is the number of elements exchanged, whereas the index pairs show the identification of the grid points in the server domain and the *in-field* grid points in the client domain.

Part A of Fig. 3 shows two model domains. The left hand side illustrates the decomposition of the server domain (blue). The server is run on 4 PEs. The decomposed model domains consist of 5x2 model grid boxes each. The

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

model domain of the client (red) was chosen to illustrate the grid association. The client model is run on 3 PEs. The decomposed domains of PE 0 and PE 1 consist of 2x1 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 client and the server domain. Table 1 gives the number (`NrEle`) and the index pairs of exchanged elements for this example[4]: The server PE 3 sends the data of four grid boxes in total. One element is sent to client PE 0 ($(1,1)_{s,PE3} \rightarrow (1,1)_{c,PE0}$), one element $(2,1)_{s,PE3}$ is sent to client PE 1 and client PE 2 gets two elements ($(1,2)_{s,PE3}$ and $(2,2)_{s,PE3}$). The client PE 2 operates on three grid boxes and needs to get this number of elements. The client grid box $(1,1)_{c,PE2}$ on PE 2 is filled by server PE 2, which sends its grid box $(5,1)_{s,PE2}$. The other two elements are sent by server PE 3: client grid box $(2,1)_{c,PE2}$ gets the data from server grid box $(1,2)_{s,PE3}$ and client grid box $(3,1)_{c,PE2}$ gets the data of server grid box $(2,2)_{s,PE3}$.

The information, which client *in-field* grid box corresponds to which server model domain grid box is saved for one client-server 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.

The structure component `NrPoints` is only relevant on the server side of the library. It is the overall number of horizontal elements the current server PE has to send to all PEs of one client. In the example this would be 0 for server PE 0, 2 for server PE 1, 1 for server PE 2 and 4 for server 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 structures of TYPE `ArrayDef`:

```
TYPE ArrayDef
   CHARACTER(LEN=STRLEN_CHANNEL)          :: channel = '' ! Name of Channel
   CHARACTER(LEN=STRLEN_OBJECT)           :: object  = '' ! Name of Object
   CHARACTER(LEN=STRLEN_OBJECT)           :: repr    = '' ! Representation
                                                          !  of Object
   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 unambiguously identification of each *exchange field*, `ArrayDef` contains the *channel* name, the *channel object* name and the *representation* of the object. `p4` is the POINTER to the respective memory, i.e., to the *in-field* on the client side and to the variable on the server side. The second block in the structure definition of `ArrayDef` contains the information about the dimensions of the array:

[4]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 c indicates server or client model, respectively, and the respective PE of the server or client 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)

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 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[5]. 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 length of the respective dimensions. The first element of `dim` gives the lebgth of the first dimension of an array, the second entry the length of the second dimension etc. An example is shown in Table 2. The structure components of `ArrayDef` discussed so far describe the properties on the current PE and thus are independent of the *remote 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 PE*, where the respective array starts.

  The buffer exchanged between one server and one client 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.

The subroutine `sort_2d_i` gets a 2-dimensional array and an index for the first rank as input. Using this index, it sorts the 2-dimensional array along its second rank from small to large values. This subroutine is used to reorder the `index_list` acoording to the server PE numbers and afterwards, for each server PE according

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

to the client numbers (see Subsect. 3.1.5).

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

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

3.1.2 `mmd_handle_communicator.f90`

The Fortran95 file `mmd_handle_communicator.f90` provides those routines required to assign each process with the communicators and MPI-rank information. These are

- the group communicator addressing all tasks of one model,
- the communicators addressing the *remote model* enabling the data exchange between the client-server model pairs.

The following subroutines are provided:

- `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_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 ist 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], 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 in 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 server and client (`m_to_client_comm`) and vice versa (`m_to_server_comm`) are determined. As a server can feed a number of clients, `m_to_client_comm` is a 1-dimensional array with the dimension `MMD_MAX_MODELL`.
∗ For clients, the server type (`m_ServerType`) is set depending on the server model name. `m_ServerType` is an INTEGER and is set to `MMD_ServerIsECHAM`, if the server name is 'echam' and to `MMD_ServerIsCOSMO`, if it is 'cosmo'. For future applications of the MMD library other PARAMETERs defining a server type can be added.
∗ The server additionally needs a list of the IDs of its clients. This information is stored in the 1-dimensional array `MMD_Server_for_Client`, which is allocated by the number of clients a specific server has to deal with.

– `MMD_get_model_communicator`:
This subroutine provides the model communicator `m_model_comm` to the basemodel calling this subroutine.

- `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.

- `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_Server_for_Client`.

- `PRIVATE read_coupling_layout`:

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

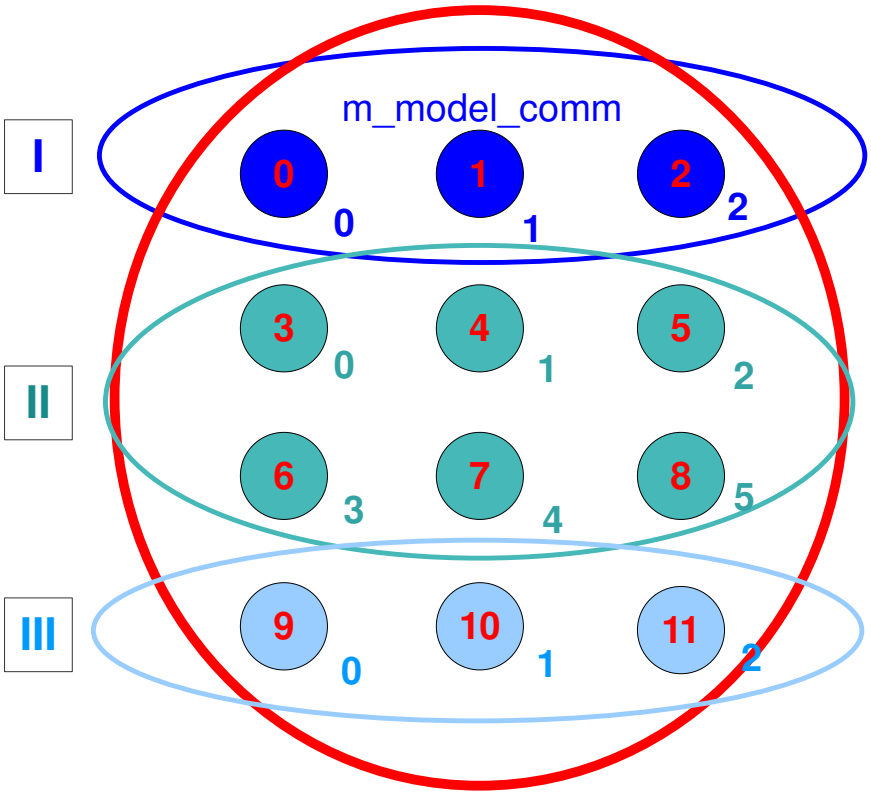

Figure 4: Illustration of the variables defined in `mmd_handle_communicator`: The small circles symbolise the individual tasks. The red circle encloses all tasks visualising the global MPI-communicator (`MPI_comm_world`). The overall number of tasks `m_world_npes` is 12 in this example. The rank of each 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 identical 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.

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 server ID of each model.

The corresponding namelist for the example illustrated in Fig. 5 with the global chemistry climate model ECHAM5/MESSy as *master server* und the regional COSMO/MESSy model as clients is given by:

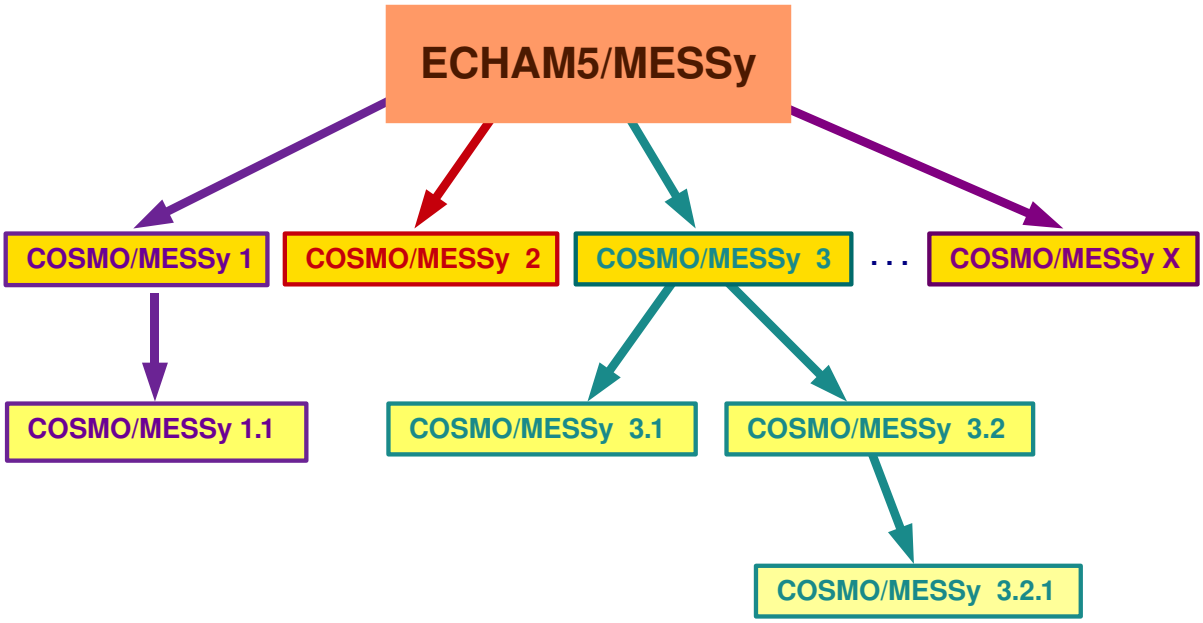

Figure 5: Example for a possible MMD model layout.

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

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 server of the respective model (the so-called `ParentId`). `-1` in the entry for 'echam' signifies that this model has no server, in other words it is the *master server*. The third column determines the number of Process Entities (PEs) required for each model[6], 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 *master server*. It is server for the four COSMO/MESSy models with the MMD internal numbers 2,3,4,5. The MMD internal model number 6 is client of the MMD internal model number 2, which is "COSMO/MESSy 1" in our example (Fig. 5). The MMD internal models number 7 and 8 are clients of the model number 4 ("COSMO/MESSy 3" in Fig. 5). Last but not least, the model with the internal number 9 is client to the model number 8 ("COSMO/MESSy 3.2").

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

[6]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`.

3.1.3 mmd_mpi_wrapper

The module mmd_mpi_wrapper provides some high level interface routines for data exchange between the client and the server model and vice versa. Send and Receive for the client and the server model are managed by four subroutines:

- MMD_Send_to_Server:

SUBROUTINE MMD_Send_to_Server		(buf, n, Server_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
Server_rank	INTEGER	IN	rank of receiving server PE in the server 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 client model and sends a data array from the client to the server.

- MMD_Recv_from_Client:

SUBROUTINE MMD_Recv_from_Client		(Client_Id, buf, n, Client_rank , tag, ierr)	
name	type	intent	description
mandatory arguments:			
Client_Id	INTEGER	IN	ID of sending client 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
Client_rank	INTEGER	IN	rank of sending client PE in the client 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 server and receives a data array from the client.

- `MMD_Recv_from_Server`:

`SUBROUTINE MMD_Recv_from_Server`		`(buf, n, Server_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
Server_rank	INTEGER	IN	rank of sending server PE in the MPI group communicator for the server
tag	INTEGER	IN	tag for data transfer to unambiguously identify this data package
ierr	INTEGER	OUT	error flag

This subroutine is called by the client and receives a data array from the server.

- `MMD_Send_to_Client`:

`SUBROUTINE MMD_Send_to_Client`		`(Client_Id, buf, n, Client_rank , tag, ierr)`	
name	type	intent	description
mandatory arguments:			
Client_Id	INTEGER	IN	ID of client 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
Client_rank	INTEGER	IN	rank of receiving client PE in the MPI group communicater for the client 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 server and sends a data array from the server to the client.

With these routines the user does not have to 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.

In addition, two broadcasting subroutines have been implemented:

- `MMD_Inter_Bcast`:

`SUBROUTINE MMD_Inter_Bcast`		`(buf [, Client_id] [, ierr])`	
name	type	intent	description
mandatory arguments:			
buf	INTEGER, DIMENSION(:)	INOUT	buffer to exchange
optional arguments:			
Client_Id	INTEGER	IN	ID of (*remote*) client (used by server only).
ierr	INTEGER	OUT	error flag

The second subroutine is called MMD_Inter_Bcast and it provides the possibility to exchange 1-dimensional INTEGER 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

First, 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_client_comm or m_to_server_comm, respectively), to both models of a client-server Pair (use the inter-communicator) or all models (set comm to the world communicator).

3.1.4 mmd_client.f90

The Multi-Model-Driver (MMD) consists of three parts. The MMD library which is discussed here, and one MESSy submodel on the client (MMDCLNT) and on the server (MMDSERV) side each. The routines included in the MMD library module mmd_client contain the client specific part of the MMD library and interact directly with the MESSy client submodel MMDCLNT. mmd_client includes eight subroutines and two functions:

- MMD_C_GetServerType:

INTEGER FUNCTION MMD_C_GetServerType	()

The INTEGER function MMD_C_GetServerType provides information about the associated server to the client submodel. It returns an INTEGER value indicating the type of the server. At the time being this is one of MMD_ServerIsECHAM or MMD_ServerIsCOSMO.

- MMD_C_Init:

SUBROUTINE MMD_C_Init	()

At the beginning of the initialisation phase the client side of the MMD library needs to be initialised. The variable Me of TYPE ClientDef is declared in the mmd_client module. In the subroutine MMD_C_Init the structure component Me%PEs is dimensioned according to the number of server PEs. All POINTERs, which are not yet ASSOCIATEd are NULLIF(Y)ied, i.e., Me%Ar, Me%ArrayStart and Me%PEs(:)%locInd. In addition, the MMD internal variable BufLen, which stores the length of the buffer received from each server PE is allocated to the number of *remote PEs*. All structure components Me%PEs(:)%NrEle and the variable BufLen(:) are initialised with zero.

- `MMD_C_Set_DataArray_Name`:

`SUBROUTINE MMD_C_Set_DataArray_Name`		`(serv_channel, serv_object, clnt_channel, clnt_object, clnt_repr, istat)`	
name	type	intent	description
mandatory arguments:			
serv_channel	CHARACTER(LEN=*)	IN	name of server *channel*
serv_object	CHARACTER(LEN=*)	IN	name of server *channel object*
clnt_channel	CHARACTER(LEN=*)	IN	name of client *channel*
clnt_object	CHARACTER(LEN=*)	IN	name of client *channel object*
clnt_repr	CHARACTER(LEN=*)	IN	*representation* of client *channel object* as given in the namelist
istat	INTEGER	OUT	error/status flag

The list of *exchange fields*, which is determined by the client submodel MMDCLNT, needs to be initialised within the MMD library. So far, MMDCLNT as client submodel reads a namelist containing a list of *exchange fields*. 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 server *channel* and *channel object* names, the client *representation* as given in the MMDCLNT namelist file and an index are broadcasted to the server.

- `MMD_C_Set_DataArray_Name_EndList`:

`SUBROUTINE MMD_C_Set_DataArray_Name_EndList`	`()`

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

- `MMD_C_Get_Indexlist`:

`SUBROUTINE MMD_C_Get_Indexlist`	`()`

The most important contribution of the server to the data exchange apart from the data itself is the list attributing the data points of the parallel decomposed server grid to the parallel decomposed client *in-field*. The server calculates the index list, which interlinks the data grid point (i_s,j_s) on server process PE_s with the client 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 server model. Each server PE sends the number of elements `NrEle`, which will be sent during the buffer exchange to the respective client 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 server and stored in `Me%PEs(ip)%locInd` (`ip` is the number of the respective server 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 client MESSy submodel MMDCLNT and the POINTER to this memory is handed to the MMD library.

The allocation of the memory within MMDCLNT is described in detail within the "MMD user manual"[7]. In order to enable the exchange of fields, which *representation* is not a priori known, the subroutine `MMD_C_Get_Repr` (in `mmd_client`) and `MMD_S_Sent_Repr` (in `mmd_server`) have been added to the MMD library to exchange the information required to determine the *representation* on the client side from the information given by the server. These routines are only called, if the *representation* in the MMDCLNT namelist file was set to `'#UNKNOWN'`. In this case the server sends the *representation* name of the respective server *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 client side within the subroutine `MMD_C_Get_Repr` and handed to the MMDCLNT 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 client submodel, the respective POINTERs can be made available to the MMD library. For this the client 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 client submodel MMDCLNT. With each call, this function internally steps one entry forward within the concatenated list.

- `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

For each of the fields the subroutine `MMD_C_Set_DataArray` is called, which associates the respective POINTER of the array and stores

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

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

- `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, a subroutine 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 allocate for the buffer exchange (see Sect. 3.2.5). The C function is called by the subroutine `MMD_C_SetInd_and_AllocMem`. 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 during the integration phase, the server writes the required data into the memory buffers made available by `MPI_alloc_mem`, which are subsequently read by the client. The subroutine `MMD_C_GetBuffer` calls for each server PE its C counterpart `MMDc_C_GetBuffer` to read the respective buffer (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 server PE into their usual 4D structure the *in-fields* of the client submodel are filled and can be processed by the client submodel afterwards. During the integration, this subroutine can be called as often as required.

- `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_server.f90`

The module `mmd_server` contains the server specific Fortran95 part of the MMD library. In contrast to the client side, the server can provide data to more than one client model. Therefore, many of the structures used on the client side must have an array dimension on the server side. In the following, two indices to identify a client model are distinguished:

- The variable `ClientId` always indicates the index of the client 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 `ClientId` depends on the entries in the MMD namelist file `MMD_layout.nml`. In `mmd_server` this index is used to address the correct client communicator (`m_to_client_comm`) and the index is parameter in 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 client models for each individual server is known. Thus, the data definition structure (and other variables the server has to provide for each client model separately) are dimensioned by the number of clients of the respective server. `Id` is the index used to address the client specific data structure on the specific server.

Hereafter, the subroutines provided by the module `mmd_server` are listed. All routines, except `MMD_S_Allocate_Client` and `MMD_S_FreeMem` are called separately for each client model. Thus one of the two indices introduced above (`ClientId` or `Id`) is always parameter of the subroutine calls to identify the current client model.

- `MMD_S_Allocate_Client`:

SUBROUTINE MMD_S_Allocate_Client			(NumClients)
name	type	intent	description
mandatory arguments:			
NumClients	INTEGER	IN	number of client models

At the very beginning, in the subroutine `MMD_S_Allocate_Client` a POINTER of TYPE `ClientDef` (named `Clients`) is allocated according to the number of clients of the specific server.

- `MMD_S_Init`:

SUBROUTINE MMD_S_Init			(ClientId, Id)
name	type	intent	description
mandatory arguments:			
ClientId	INTEGER	IN	index of client model within the overall MMD model setup: this subroutine is called separately for each client of the server.
Id	INTEGER	IN	index of client model in the client list of this specific server

First, the server needs to be initialised for each client model. The subroutine `MMD_S_Init` inquires the number of processes occupied by the respective client (`Clients(Id)%inter_npes`) and allocates `Clients(Id)%PEs` accordingly. `Clients(Id)%PEs(ip)%NrEle` is initialised with 0 (`ip` is the index of the *remote PE*) and `Clients(Id)%PEs(ip)%locInd` is NULLIF(Y)ied.

- `MMD_S_Set_Indexlist`:

SUBROUTINE MMD_S_Set_Indexlist			(Id, index_list)
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server
index_list	INTEGER, DIMENSION(:,:)	INOUT	index list (used for the horizontal element association) as calculated from the server submodel

The most tricky part of the MMD library is the association of grid points from the parallel decomposed server field with the grid points on the parallel decomposed client *in-fields*. For that, the server submodel MMDSERV receives the geographical longitude and the geographical latitude fields of the parallel decomposed client "in"-grid[8], 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 server submodel identifies for each of the client grid points the respective grid point in the local server model domain, thus adding to the list the server process number PE_s (of the model specific MPI-communicator) on which the respective geographical point is located and the respective index pair (i_s,j_s) in the local domain. Thus a list of n sextuples (i_s,j_s,i_c,j_c,PE_c, PE_s)$_n$ containing the index pairs in both decomposed fields and the client and the server PE number is created, with n being the overall number of exchanged horizontal elements.

[8] As the server and the client "in"-grid can be a rotated grid, the geographical longitudes and latitude fields are 2D fields each.

This list is further analysed within the MMD library routine `MMD_S_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 server PE numbers and calculates the number of grid points each server PE has to sent (`Clients(Id)%NrPoints`). This number is sent to the respective server PE. Additionally, the part of the index list of the respective server PE is sent to it. Each server PE deduces from its part of the index list the number of horizontal elements `Clients(Id)%PEs(ip)%NrEle` it has to send to each individual client PE. After that, the index pairs of the local server grid associated with the horizontal elements are saved in the `locInd` structure `Clients(Id)%PEs(ip)%locInd` and the index pairs of the local client grid are saved in an intermediate variable. In the following the number of horizontal elements and the intermediate variables are exchanged with the respective client 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_S_Get_DataArray_Name`:

`SUBROUTINE MMD_S_Get_DataArray_Name (Id)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server

For the coupling of the data fields, the *channel* and *channel object* names of the *exchange fields* and the *representation* name as provided by the client model must be received by the server from each client. Within the subroutine `MMD_S_Get_DataArray_Name` the *channel* and *channel object* names and the *representation* name are acquired from the client 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 `ClientDef` structure) defining the individual data fields is established. The memory location of the first array is stored in the variable `Clients(Id)%ArrayStart`. The server *channel* and *channel object* name and the client *representation* name are stored in the structure components `Clients(Id)%Ar%ArrDef%channel`, `Clients(Id)%Ar%ArrDef%object` and `Clients(Id)%Ar%ArrDef%repr`, respectively, making these strings available for later use on the server side of the MMD library.

- `MMD_S_GetNextArray`:

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

So far, the *channel* and *channel object* names are only known within the MMD library. For the acquisition of the server fields, they must be made available for the server submodel MMDSERV. With each call to the function `MMD_S_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 client model are forwarded to the server submodel. The server submodel obtains the requested data POINTER based on the *channel* and *channel object* name.

- `MMD_S_Send_Repr`:

`SUBROUTINE MMD_S_Send_Repr`		`(axis, gdimlen, name, att, ClientId)`	
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
ClientId	INTEGER	IN	index of client in the MMD model setup

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

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

to the client.

- `MMD_S_Set_DataArray`:

`SUBROUTINE MMD_S_Set_DataArray`		`(Id, status, DIMLEN, ArrayOrder, p4)`	
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server
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*

Finally, the POINTER to the memory of the server *exchange fields* are passed to the MMD library by the subroutine `MMD_S_Set_DataArray` and saved in the structure component `Clients(Id)%Ar%Arrdef%p4`. Additionally,

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

are stored within the `Clients` structure. Making use of this information, the individual array length (`Clients(Id)%Ar%Arrdef%ArrLen(ip)`) and the length of the buffer sent by each server PE to each client PE are calculated (`BufLen(ip)`).

- `MMD_S_SetInd_and_AllocMem`:

`SUBROUTINE MMD_S_SetInd_and_AllocMem`		`(Id)`	
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server

After all POINTERs and dimension information are stored, the full buffer size is determined within the C part of the library as sum over all buffer lengths (`SUM(BufLen)`) sent to each individual client PE. This is triggered by the call of the subroutine `MMDc_S_SetInd_and_Mem`. As the full size of the buffer is only required in the C part of the library the Fortran95 subroutine `MMD_S_SetInd_and_AllocMem` simply calls the C function `MMDc_S_SetInd_and_Mem`. After processing this subroutine the initialisation is finished.

- `MMD_S_FillBuffer`:

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

Within the time loop the subroutine `MMD_S_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 server arrays is not a priori known on the client side, the packing algorithm in `MMD_S_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_S_FillBuffer` contains an algorithm packing an array of arbitrary order of *dimensions* in grid point *representation*[9]. 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_S_FillBuffer` is called, copying the array to the memory space allocated by `MPI_Alloc_mem`. When all buffers attributed to all client PEs are filled, the C function `MMDc_S_BufferFull` is called, which sets a barrier to prevent the buffer to be filled a second time before the data was read by the client.

- `MMD_S_FreeMem`:

`SUBROUTINE MMD_S_FreeMem (NumClients)`			
name	type	intent	description
mandatory arguments:			
NumClients	INTEGER	IN	number of client models of this specific server

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

3.1.6 `mmd_test.f90`

The most tricky parts of the data exchange are to match the packing algorithms of the server and the client and the creation of the `index_list`, in which the index pair of each horizontal element of the client *in-field* on each client PE is associated with the index pair of the associated horizontal element in the local server domain and the respective PE number (in the server 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 server side is a three dimensional field spanned by a 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.

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

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

N	field
1	Geographical longitude as defined for the server grid
2	Geographical latitude as defined for the server grid
3	First index in parallel decomposed server grid (i_s)
4	Second index in parallel decomposed server grid (j_s)
5	First index in parallel decomposed client grid (i_c)
6	Second index in parallel decomposed client grid (j_c)
7	Number of respective client process (PE_c)
8	Number of respective server process (PE_s)

During the exchange procedure each horizonal grid point (and the associated eight entries) is assigned to a grid point of the *in-field* of the client. Thus the longitude and the latitude of each grid point known to the client for the *in-fields* can directly be compared with the longitude and the latitude in the exchanged server 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 client 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 server and client submodels.

- `MMD_testC_Setup`/`MMD_testS_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_testS_Setup (Id, nx, ny, cdim_order)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server
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 client model (subroutine `MMD_testC_Setup`) as the `test_array` is simply allocated according to the local horizontal dimensions (as provided by the client submodel) and the required additional "numerical" dimensions of the array on each client PE. The setup of the server (in `MMD_testS_Setup`) is more demanding. First, the local dimensions are not necessarily equal on all server PEs. Therefore the maximum local horizontal dimension length of all server 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'`)[10] have to be distinguished. Thus, more than one `test_array` is required during the initialisation phase of the server:

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

  - 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_testS_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.
  - 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 server PEs) horizontal dimensions.

- `MMD_testS_GetTestPtr/MMD_testC_GetTestPtr`:

`SUBROUTINE MMD_testS_GetTestPtr (Id, p, axis, ldim)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server
p	REAL(dp),DIMENSION(:,:,:,:)	POINTER	Return 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	return 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 client and the server submodel using the subroutines `MMD_testS_GetTestPtr` and `MMD_testC_GetTestPtr`, respectively.

- `MMD_testS_Fill`:

`SUBROUTINE MMD_testS_Fill (Id, index_list, lon, lat)`			
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server
index_list	INTEGER, DIMENSION(6)	IN	sextuple describing the connection between one grid point on one server PE with one grid point of a client PE; This list contains: index pair on server PE, index pair on client PE, number of client PE, number of server 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 server and client required similar preparations. The rest of the initialisation phase is purely work on the server side as the `test_array` needs to be filled with the `index_list` information. The subroutine `MMD_testS_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_testS_FinishFill`:

`SUBROUTINE MMD_testS_FinishFill`		`(Id)`	
name	type	intent	description
mandatory arguments:			
Id	INTEGER	IN	index of client model in the client list of this specific server

At the end of the filling procedure within the subroutine `MMD_testS_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 client PEs received all buffers from all server 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_testS_FreeMem`:

`SUBROUTINE MMD_testS_FreeMem`		`(ClientId)`	
name	type	intent	description
mandatory arguments:			
ClientId	INTEGER	IN	index of client 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_testS_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 server and client submodels (MMDCLNT and MMDSERV) should be easily expandable for a scientific user. Even more important: the data fields filled on the client side are accessed by Fortran95 POINTERs. These POINTERs can point to non-contiguous sub-samples of higher dimensioned TARGETs. In contrast, C pointers provide access to a field by pointing to the memory address, where this field starts assume that the field is stored contiguously in the memory. The MMDCLNT submodel provides 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 omitted 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 required to enable the data exchange (e.g. parameter lists) between Fortran95 and C routines.

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 communictor 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 client and the server model.
- The structure `BufDef` contains all information specific for one *remote model PE*:

  ```
  struct BufDef {
    int                  BufLen;       /* Length of 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*,
  - `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 consists of 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_server.c

mmd_c_server.c contains five functions:

- MMDc_S_Init:

void MMDc_S_Init (model_comm, inter_comm, *npes)			
name	type	intent	description
arguments:			
ClientId	int	IN	index of client 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_S_Init. Additionally, this function provides the number of *remote PEs* to the Fortran95 part of the library.

- MMDc_S_SetInd_and_Me:

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

  The server 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 server submodel MMDSERV. The Fortran95 part of the MMD library calculates the dependencies between the parallel decomposed server and client grids. From this, the length of the buffer each server PE exchanges with each client PE is calculated. This is the only information required by the C part of the library.

  - The function MMDc_S_SetInd_and_Mem stores this information within the structure component &Clients[*ClientId-1]->buf->BufLen.
  - The total buffer length (&Clients[*ClientId-1]->TotalBufferSize) is calculated by summing up the individual buffer lengths exchanged with the *remote PEs*.
  - The buffers exchanged with each client PE are aligned to one large one-dimensional buffer. The information, at which index in this large buffer the array for the individual client PEs starts, is stored in the structure component &Clients[*ClientId-1]->buf->BufIndex.
  - Each of these indices is sent to the respective client PE, to enable the client PE to locate the corresponding buffer part.
  - The total buffer length (&Clients[*ClientId-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 &Clients[*ClientId-1]->BufWin.
  - Afterwards the pointers to the starting addresses of each buffer exchanged with every client PE (&Clients[*ClientId-1]->buf->SendBuf) is stored for later use.

- `MMDc_S_FillBuffer`:

`void MMDc_S_FillBuffer (ClientId, *PeId,*array )`			
name	type	intent	description
arguments:			
ClientId	int	IN	index of client 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 these pointers are used to fill the buffer. In the function `MMDc_S_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 server and client).

- `MMDc_S_BufferFull`:

`void MMDc_S_BufferFull (ClientId)`			
name	type	intent	description
arguments:			
ClientId	int	IN	index of client in the MMD model setup

  When the buffer is completely filled, i.e., each server PE filled the buffers for all client PEs, a barrier is set within the function `MMDc_S_BufferFull` to prevent the server from overwriting the buffer before it is copied by the client. After the client read the data, the barrier is released by the client and the server can fill the buffer again.

- `MMDc_S_GetWaitTime`:

`void MMDc_S_GetWaitTime (ClientId,*WaitTime )`			
name	type	intent	description
arguments:			
ClientId	int	IN	index of client in the MMD model setup
WaitTime	double	OUT	time span the server had to wait

  `mmdc_server.c` comprises the additional function `MMDc_S_GetWaitTime`. It measures the time span between the call of the FillBuffer function and the release of the buffer by the client.

3.2.5 `mmdc_client.c`

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

1. The initialisation of the MMD library client 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 client 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 client 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 server model (`model_npes` and `inter_npes`) are determined. Based on the number of server PEs, the variable `buf` of type `BufDef` is allocated. The number of server 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 server PE to the respective client PE.

`void MMDc_C_SetInd_and_Mem`		`(*bufsize)`	
name	type	intent	description
arguments:			
bufsize	int, DIMENSION(:)	IN	length of buffer exchange per *remote PE*

In function `MMDc_C_SetInd_and_Mem` the information about the length and the location of the buffer sections accessed by each specific client PE within the exchanged buffers is stored.

- The calling Fortran95 subroutine provides the buffer length, which is required as input and independent for each of the server PEs by the current client PE. This length is stored in the structure component `me->buf->BufLen`.
- The server PE sends an index, which indicates, where the data for the client PE starts in the 1-dimensional array of data sent by the server PE. This index is stored in the structure component `me->buf->BufIndex`.
- Additionally, the maximum length of all buffer lengths exchanged by the current client PE (`maxbufsize`) is determined. This is the required buffer size on the client side, because the client exchanges data only with one server PE at once.
- Thus, a buffer of respective size is allocate with `MPI_alloc_mem`.
- For accessing the data provided by the server 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 server PE is stored in `me->buf->SendBuf`.

3. The actual buffer exchange:

`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 client submodel.

- `MMDc_C_GetWaitTime`:

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

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

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.

Figure 6 illustrates the buffer definition on the server side in the C part of the library. Each grey box depicts one server PE. The first grey box is empty, as the client domain does not overlap with PE_s 0. Server PE_s 1 contributes one grid point to client PE_c 0 and one to PE_c 1. This is illustrated by the coloured bars: yellow

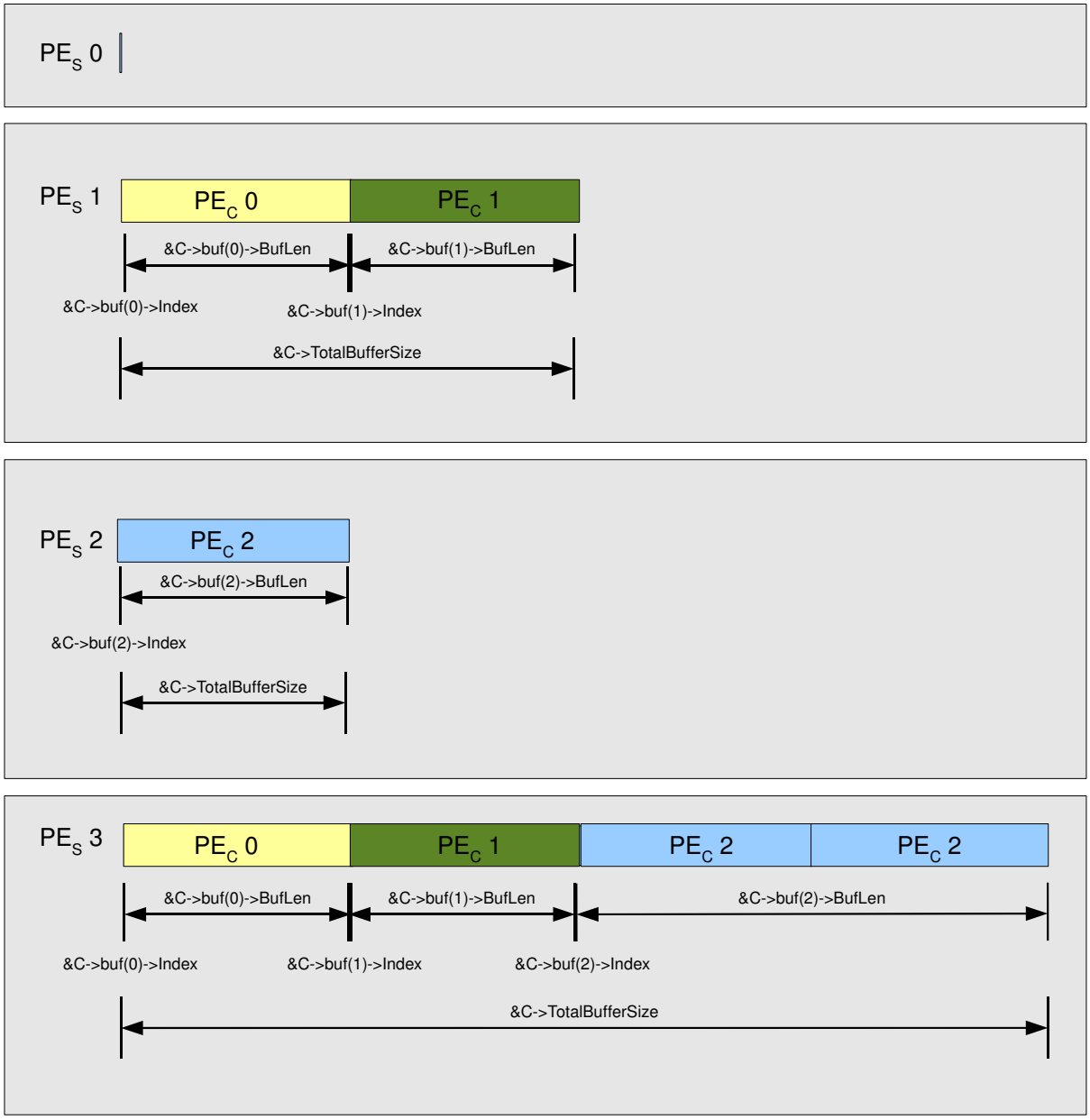

Figure 6: Example of a possible MMD exchange buffer layout.

for client PE$_c$ 0 and green for PE$_c$ 1. Each of this buffer parts is `&Clients[*ClientId-1]->buf->BufLen` long. In the figure the first part of the structure `&Clients[*ClientId-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. , `&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 client PEs starts. This is indicated by the respective indices `&C->buf( )->Index`. The overall length of the buffer sent by PE$_s$ 1 is given by `&C->TotalBufferSize`.

Server PE$_s$ 2 only sends data to client PE$_c$ 2. Thus `&C->TotalBufferSize` for PE$_s$ 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 client PEs are set to zero.

Server PE$_s$ 3 sends data to all three client PEs: one grid point each to client PE$_c$ 0 and PE$_c$ 1 and two grid

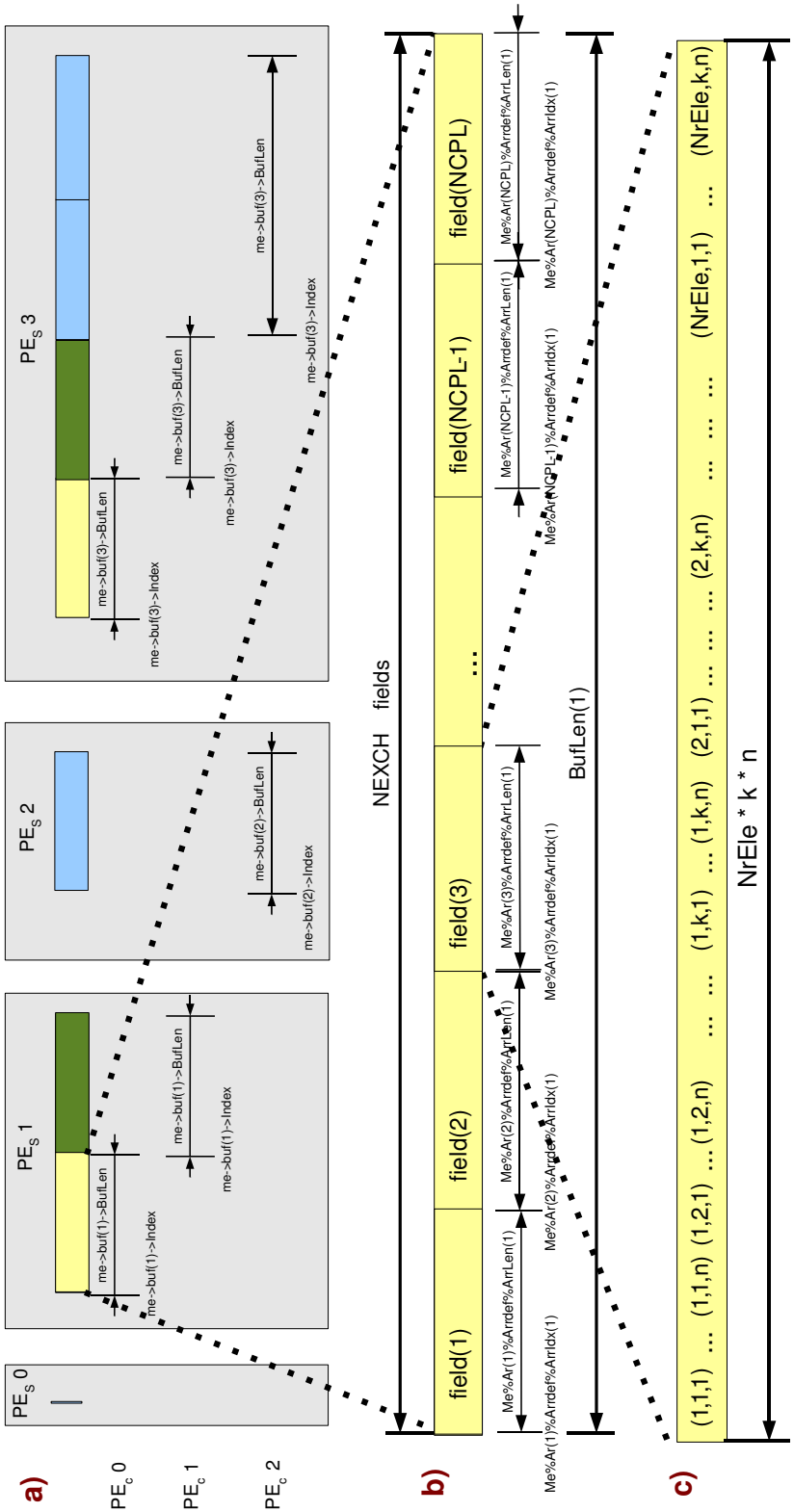

Figure 7: Example of a possible MMD exchange buffer layout.

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 client side. The figure is constructed like a table. The server PEs build the columns and the client PEs the rows. The grey boxes indicate again the server PEs. As PE_s 0 does not send any data, all variables are zero on all client PEs. client PE_c 0 receives data from PE_s 1 and PE_s 3. The buffer sent by PE_s 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_s 1. Correspondingly, the buffer received from PE_s 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 client PE_c 1 are defined accordingly. client PE_c 2 receives data from the two server processes PE_s 2 and PE_s 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_s 3 but only one from PE_s 2.

The C part needs to deal only with the complete buffers exchanged between one server and one client 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 client and per server 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[11]. The individual length of each field is saved in the variable `Me%Ar(ix)%Arrdef%ArrLen(ip)`. Where `ix` is the number of the field. This is again a pseudo-index used for illustration, as `Me%Ar` is a concatenated list. The index `ip` indicates the respective server PE. `Me%Ar(ix)%Arrdef%ArrLen(ip)` can be different for each server PE, because the number of elements exchanged with each server PE (`Me%PEs(ip)%NrEle`) can be different for each server 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_client.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 *channel object*, 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*.
- *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 `mmdclnt.nml` namelist file and provided by the server to the client. An *exchange field* can either be a field which is interpolated and copied to a client variable, or a field required for the interpolation itself.
- *in-field*: The *in-fields* are those fields provided by the server or *driving model*, which are still defined on the server grid, but on the client side. In other words, *in-fields* are the *exchanged fields* before the interpolation.

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

- *master server*: The *master server* is the coarsest model in a model cascade, i.e., that model that has no server itself. In the MMD library namelist this model is indicated by a "-1" as associated server. In most cases this is a global model. The *master server* determines the timing of the entire model cascade.
- *remote model*: the "other" model in a communicating client-server pair; i.e., for the client the server, for the server the respective client
- *remote PE*: the "other" PE in a pair of client and server PEs exchanging data. For example, server PE_s 2 is sending data to client PE_c 4: in this case PE_c 4 is the *remote PE* for server PE_s 2 and vice versa.
- *representation*: It describes multidimensional geometric structures (based on *dimensions*), e.g., Eulerian (or grid point), spectral, Lagrangian.

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.