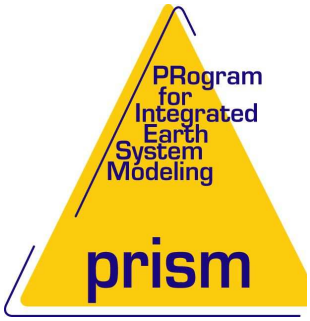
PRISM
A Software Infrastructure Project for Climate Research in Europe



OASIS4 User Guide
(OASIS4_0_2)

Edited by:
S. Valcke, CERFACS
R. Redler, NEC-CCRLE

PRISM–Support Initiative
Technical Report No 4

CERFACS TR/CMGC/06/74
August 25, 2006

Copyright Notice

How to get assistance?

Assistance can be obtained by sending an electronic mail to oasis4_help(at)lists.enes.org and as listed in Contact below.
PRISM documentations can be downloaded from the WWW PRISM web site under the URL: <http://prism.enes.org>

Contacts

Name	Phone	Affiliation
Sophie Valcke	+33-5-61-19-30-76	CERFACS
René Redler	+49-2241-92-52-40	NEC-CCRLE

Contents

Chapter 1

Introduction

A new fully parallel coupler for Earth System Models (ESMs), OASIS4, has been developed within the European PRISM project. Chapter 2 provides a more detailed description of OASIS4 sources available from CERFACS CVS server.

An ESM coupled by OASIS4 consist of different applications (or executables), which executions are controlled by OASIS4. Each ESM application may host only one or more than one climate component models (e.g. model of the ocean, sea-ice, atmosphere, etc.). To interact with the rest of the ESM at run-time, the component models have to include specific calls to the OASIS4 PRISM System Model Interface Library (`PSMILe`). Each application and component model must be provided with XML files (10) that describe its coupling interface established through `PSMILe` calls. The configuration of one particular coupled ESM simulation, i.e. the coupling and I/O exchanges that will be performed at run-time between the components or between the components and disk files, is done by the user also through XML files.

During the run, OASIS4 Driver/Transformer's role is to extract the configuration information defined by the user in the XML files, to organize the process management of the coupled simulation, and to perform the regridding needed to express, on the grid of the target models, the coupling fields provided by the source models on their grid. The OASIS4 Driver/Transformer is described in chapter 3.

The `PSMILe` , linked to the component models, includes a data exchange library which performs the MPI-based (Message Passing Interface) (9) exchanges of coupling data, either directly or via additional Transformer processes, and the GFDL mpp_io library (2), which reads/writes the I/O data from/to files following the NetCDF format (8). The `PSMILe` and its Application Programming Interface (API) are described in chapter 4.

The structure and content of the descriptive and configuring XML files are then detailed in chapter 5. In chapter 6, instructions on how to compile and run the example toy coupled model TOYOA4 using OASIS4 are given; a toy model is an empty model in the sense that it contains no physics or dynamics; it reproduces, however, a realistic coupling in terms of number of component models, number, size and interpolation of the coupling or I/O fields, coupling or I/O frequencies, etc. Results of the OASIS4 scalability test performed in 2004, at the end of the EU PRISM project funded by the European Community are finally given in appendix A, even if they were not performed with the current OASIS4 version.

During the last year, OASIS4 started to be used into real applications by the GEMS community (ECMWF, Météo-France, and KNMI) for 3D coupling between atmospheric dynamic and atmospheric chemistry models, by SMHI for ocean-atmosphere regional coupling, and by the UK MetOffice for global ocean-atmosphere coupling.

Other MPI-based parallel coupler performing field transformation exist, such as the 'Mesh based parallel Code Coupling (MpCCI)' (1) or the 'CCSM Coupler 6' (3). The originality of OASIS4 relies in its great flexibility (as the coupling and I/O configuration is externally defined by the user in XML files) in its parallel neighborhood search based on the geographical description of the process local domains, and in its common treatment of coupling and I/O exchanges, both performed by the `PSMILe` library.

Chapter 2

OASIS4 sources

2.1 Copyright Notice

This software and ancillary information called OASIS4 is free software. The public may copy, distribute, use, prepare derivative works and publicly display OASIS4 under the terms of the Lesser GNU General Public License (LGPL) as published by the Free Software Foundation, provided that this notice and any statement of authorship are reproduced on all copies. If OASIS4 is modified to produce derivative works, such modified software should be clearly marked, so as not to confuse it with the current OASIS4 version.

The developers of the OASIS4 software attempt to build a parallel, modular, and user-friendly coupler accessible to the climate modelling community. Although we use the tool ourselves and have made every effort to ensure its accuracy, we can not make any guarantees. The software is provided for free; in return, the user assume full responsibility for use of the software. The OASIS4 software comes without any warranties (implied or expressed) and is not guaranteed to work for you or on your computer. CERFACS, NEC-CCRLE, SGI Germany, NEC HPCE, and CNRS and the various individuals involved in development and maintenance of the OASIS4 software are not responsible for any damage that may result from correct or incorrect use of this software.

2.2 Reference

If you feel that your research has benefited from the use of the OASIS4 software, we will greatly appreciate your reference to the following report:

Valcke, S., R. Redler, 2006: OASIS4 User Guide (OASIS4_0_2). PRISM Report No 3, 2nd Ed., 60 pp.

2.3 How to obtain OASIS4 sources

OASIS4 sources, Makefiles, and toy example can be retrieved from the CERFACS CVS server *alter* or from CERFACS anonymous ftp. The functionality described in this report correspond to the sources tagged "OASIS4_0_2". For more detail on how to obtain the sources, please contact us.

2.4 OASIS4 directory structure

2.4.1 OASIS4 sources

OASIS4 sources were divided into three directories under `prism/src/lib/` and one directory `prism/src/mod/oasis4/`. In this new structure, only a relatively small library *common_oa4* is used by both the OASIS4 Driver/Transformer executable, which at run time performs the interpolations, and

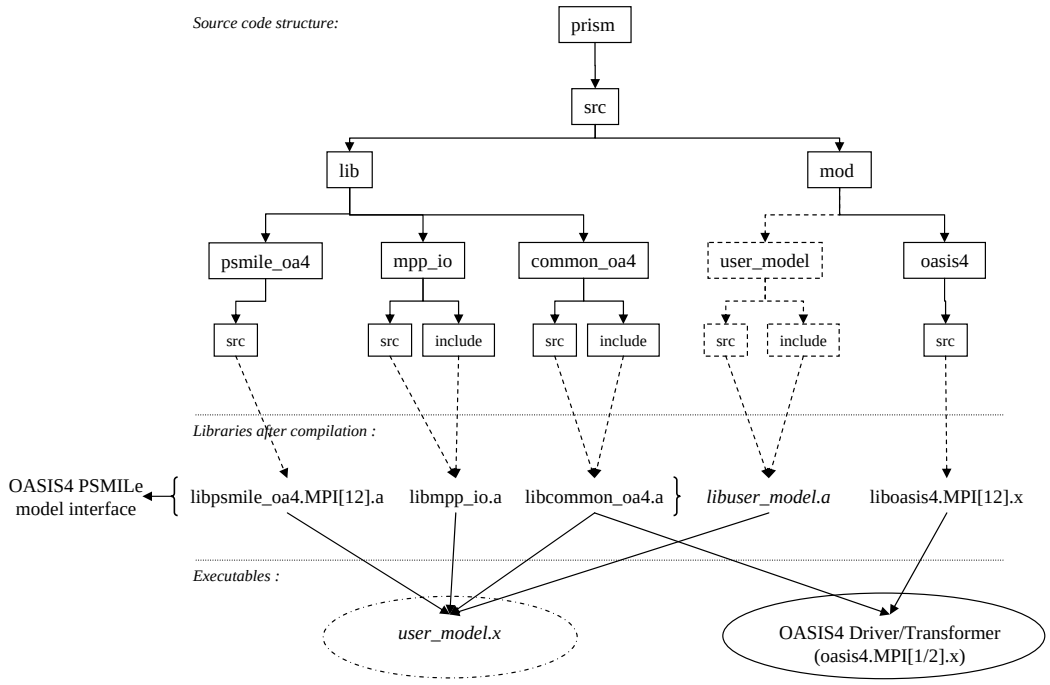


Figure 2.1: New source directory structure

by the OASIS4 PSMILe model interface, which needs to be linked to the component models for I/O and coupling exchanges. The different directories are:

- `prism/src/lib/common_oa4/`: this directory contains sources that are used both by the Driver/Transformer and the PSMILe model interface. After compilation, these sources becomes the *libcommon_oa4.a* library.
- `prism/src/lib/mpp_io/`: this directory contains the sources of the GFDL I/O library (2). After compilation, these sources form the library *libmpp_io.a*. Compiling and linking of this library to a component model is not mandatory if the PSMIle I/O functionality is not used (see compilation details in section 6).
- `prism/src/lib/psmile_oa4/`: this directory contains the sources that form the main part of PSMILe model interface and become, after compilation the library *libpsmile_oa4.a*.
- `prism/src/mod/oasis4/`: this directory contains the main part of OASIS4 Driver/Transformer sources. Linked with the library *libcommon_oa4.a*, these sources form, after compilation, the OASIS4 Driver/Transformer executable named `oasis4.MPI1.x` or `oasis4.MPI2.x` (according to the choice of MPI1 or MPI2 done at compilation, see section 6 for details).

2.4.2 Other OASIS4 directories

In the `prism/src/mod/oasis4` directory, three more directories `/doc`, `/examples` and `/util` are found:

- `/doc` contains OASIS4 documentation.

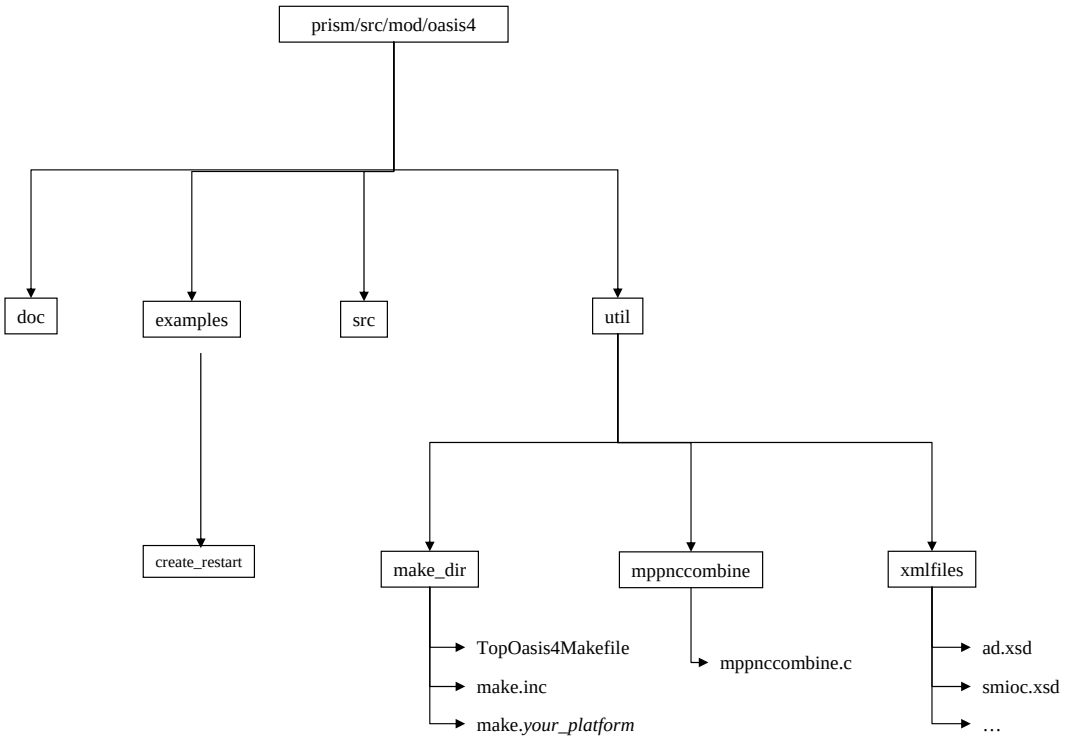


Figure 2.2: Directories in prism/src/mod/oasis4

- `/examples` and its sub-directory `/create_restart` contains programs which provide example on how to use the PSMILe `prism_put_restart` routine to create an OASIS4 coupling restart file (see the README therein).
- `/util` contains directory `/make_dir` into which a top makefile and platform dependent header files for compiling OASIS4 without using the SCE can be found (see section 6.2.1), directory `/xmlfiles` which contains the SCHEMAs of the different XML files used with OASIS4 (see section 5), and directory `mppnccombine` which contains a program, `mppnccombine.nc`, which may be used to join together NetCDF data files representing a decomposed domain into a unified NetCDF file.

2.4.3 The toy coupled model TOYOA4 directory structure

TOYOA4 provides a practical example on how to use OASIS4 to couple 3 component models. The sources for each toy component model are included in the PRISM directory structure with one directory for each component, respectively in `/prism/src/mod/atmoa4`, `/oceoa4`, and `/lanoa4`. Section 6.2 details how to compile those three toy component models while section 6.3 explains how to run the resulting toy coupled model TOYOA4.

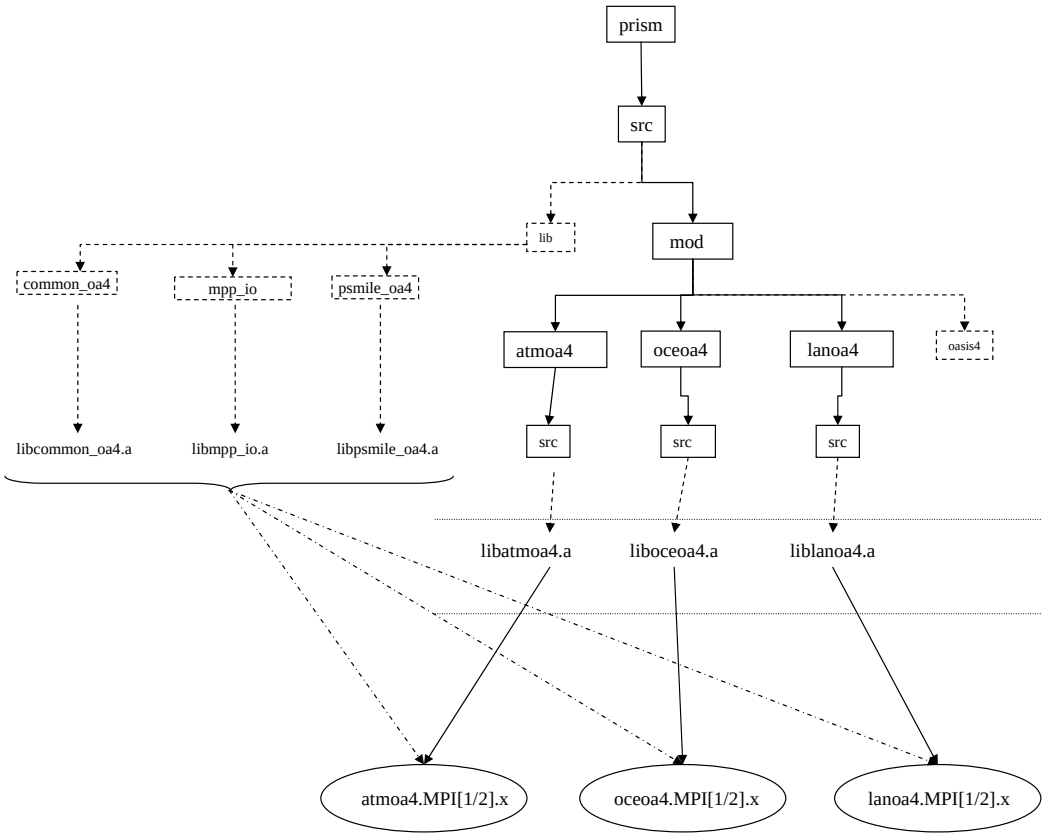


Figure 2.3: Directory structure for TOYOA4

Chapter 3

OASIS4 Driver/Transformer

OASIS4 Driver/Transformer tasks are described in this chapter to give the user a complete understanding of OASIS4 functionality. The realisation of these tasks at run-time is however completely automatic and transparent for the user. OASIS4 Driver/Transformer is parallel, although only the main process is used to execute the Driver's tasks.

3.1 The Driver part

The first task of the Driver is to get the process management information defined by the user in the SCC XML file (see section 5.4). The information is first extracted using the libxml C library (11), and then passed from C to Fortran to fill up the Driver structures.

Once the Driver has accessed the SCC XML file information, it will, if the user has chosen the `spawn` approach, launch the different executables (or applications) that compose the coupled model, following the information given in the SCC file. For the `spawn` approach, only the Driver should therefore be started and a full MPI2 implementation (5) is required as the Driver uses the MPI2 `MPI_Comm_Spawn_Multiple` functionality. If only MPI1 implementation is available (9), the Driver and the applications must be all started at once in the run script; this is the so-called `not_spawn` approach. The advantage of the `spawn` approach is that each application keeps its own internal communication context (e.g. for internal parallelisation) unchanged as in the standalone mode, whereas in the `not_spawn` approach, OASIS4 has to recreate an application communicator that must be used by the application for its own internal parallelisation. Of course, the `not_spawn` is also possible if an MPI2 library is used.

The Driver then participates in the establishment of the different MPI communicators (see section 4.1.3), and transfers the relevant SCC information to the different component model `PSMILe`s (corresponding to their `prism_init` call, see section 4.1.1).

When the PRISM simulation context is set, the Driver accesses the SMIOCs XML files information (see section 5.5), which mainly defines all coupling and I/O exchanges (e.g. source or target components or files, local transformations, etc.). The Driver sorts this component specific information, and defines global identifiers for the components, their grids, their coupling/IO fields, etc. to ensure global consistency between the different processes participating in the coupling. Finally, the Driver sends to each component `PSMILe` the information relevant for its coupling or I/O exchanges (e.g. source or target components or files and their global identifier) and information about the transformations required for the different coupling fields. This corresponds to the component `PSMILe prism_init_comp` call (see section 4.1.2)[1]. With such information, the PRISM applications and components are able to run without any other interactions with the Driver. Analysing the XML information, the PRISM Driver is able to determine how many

[1] If the component is running stand-alone, i.e. without a Driver, the `PSMILe` component automatically reads its SMIOC information below the `prism_init_comp` call. In this case, the component SMIOC is used to configures the I/O of the component from/to files.

Transformer processes are specified, if any. The Driver processes are then used to execute the Transformer routines (see Section 3.2).

When a component reaches the end of its execution, its processes send a signal to the Transformer instance by calling the PRISM_Terminate routine (see Section 4.7.1). Once the Transformer instance has received as many signals as processes active in the coupled run, the Transformer routines stop and the Driver finalizes the simulation.

3.2 The Transformer part

The PRISM Transformer manages the regridding (also called the interpolation) of the coupling fields, i.e. the expression on the target component model grid of a coupling field given by a source component model on its grid. The Transformer performs only the weights calculation and the regridding *per se*. As explained in section 4.5.1, the neighborhood search, i.e. the determination for each target point of the source points that will contribute to the calculation of its regridded value, is performed in parallel in the source `PSMILe`.

The PRISM Transformer can be assimilated to an automate that reacts following predefined sequences of actions considering what is demanded. The implementation of the Transformer is based on a loop over the receptions of predefined arrays of ten Integers sent by the component `PSMILe` . These ten integers give a clear description of what has to be done by the Transformer. The Transformer is thus able to react with a pre-defined sequence of actions matching the corresponding sequence activated on the sender side.

The first type of action that can be requested by the component `PSMILe` is to receive the grid information resulting of the different neighbouring searches. The Transformer receives, for each intersection of source and target process calculated by the `PSMILe` , the latitude, longitude, mask, or areas of all source and target grid points in the intersection involved in the regridding (EPIOS and EPIOT, see section 4.5.1). The Transformer then calculates the weight corresponding to each source neighbour depending on the regridding method chosen by the user. The end of this phase corresponds in the component models to the `PSMILe` routine `prism_enddef`.

During the simulation timestepping, the Transformer receives orders from the `PSMILe` linked to the different component processes to receive data for transformation (source component process) or to send transformed data (target component process). After a reception, the Transformer applies the appropriate transformations or regridding following the information collected during the initialisation phase (here, the regridding corresponds to applying the pre-calculated weights to the source field). In case of request of fields, the Transformer is able to control if the requested field has already been received and transformed. If so, the data field is sent; if not, the data field will be sent as soon as it will have been received and treated.

At the end of the run, the Transformer is informed by the participating processes once they are ready to finish the coupled simulation; the Transformer then gives the hand back to the Driver.

Chapter 4

OASIS4 Model Interface library, PSMILe

To communicate with the rest of the coupled system, each component model needs to perform appropriate calls to the PRISM System Model Interface Library (PSMILe). The PSMILe is the software layer that manages the coupling data flow between any two (possibly parallel) component models, directly or via additional Transformer processes, and handles data I/O from/to files.

The PSMILe is layered, and while it is not designed to handle the component internal communication, it completely manages the communication to other model components and the details of the I/O file access. The detailed communication patterns among the possibly parallel component models are established by the PSMILe. They are based on the source and target components identified for each coupling exchange by the user in the SMIOC XML files (see section 5.5) and on the local domain covered by each component process. This complexity is hidden from the component codes as well as the exchanges of coupling fields *per se* built on top of MPI. In order to minimize communication, the PSMILe also includes some local transformations on the coupling fields, like accumulation, averaging, gathering or scattering, and performs the required transformation locally before the exchange with other components of the PRISM system.

The interface was designed to keep modifications of the model codes at a minimum when implementing the API. Some complexity arises however in the API from the need to transfer not only the coupling data but also the meta-data as will be explained below.

In order to match the data structures of the various component codes (in particular for the geographical information) as closely as possible, Fortran90 overloading is used. All grid description and field arrays provided by the component code through the PSMILe API (e.g. the grid point location through prism_set_points , see 4.3.6) can have one, two or three numerical dimensions and can be of type "Real" or "Double precision". There is no need to copy the data arrays prior to the PSMILe API call in order to match some predefined internal PSMILe shape. To interpret the received array correctly, a properly defined grid type is required (see section 4.3.1), since the grid type implicitly specifies the shape of the data arrays passed to the PSMILe.

A major principle followed throughout the declaration phase and during the transmission of transient fields is that of using identifiers (ID) to data objects accessible in the user space once they have been declared. Like in MPI, the memory that is used for storing internal representations of various data objects is not directly accessible to the user, and the objects are accessed via their ID. Those IDs are of type INTEGER and represent an index in a table of the respective objects. The object and its associated ID are significant only on the process where it was created.

The PSMILe API routines that are defined and implemented are not subject to modifications between the different versions of the PRISM coupler. However new routines may be added in the future to support new functionality. In addition to that the PSMILe is extendable to new types of coupling data and grids.

The next sections describe the functioning of the PSMILe, and explain its different routines in the logical order in which they should be called in a component model.

4.1 Initialisation phase

The developer first has to use in his code the PRISM module (`'use PRISM'`, see `prism/src/lib/psmile_oa4/src/prism.F90`), which declares all PRISM structures and PRISM integer named parameters from `prism/src/lib/common_oa4/include/prism.inc` (data types, grid types, error codes, etc.). The following routines then participate in the coupling initialisation phase:

4.1.1 prism_init

`prism_init (appl_name, ierror)`

Argument	Intent	Type	Definition
`appl_name`	`In`	`character(len=*)`	name of application in SCC XML file
`ierror`	`Out`	`Integer`	returned error code

Table 4.1: prism_init arguments

The initialisation of the PRISM interface and the coupling environment is performed with a call to `prism_init`. This routine belongs to the class of so-called collective calls and therefore has to be called once initially by each process of each application, either directly or indirectly via `prism_init_comp` (see 4.1.2).

Since all communication is built on MPI routines, the initialisation of the MPI library is checked below `prism_init`, and a call to `MPI_Init` is performed if it has not been called already by the application. It is therefore not allowed to place a call to `MPI_Init` after the `prism_init` call in the application code, since this will lead to a runtime error with most MPI implementations. Conversely, a call to `prism_terminate` (see 4.7.1) will terminate the coupling. If `MPI_Init` has been called before `prism_init`, internal message passing within the application is still possible after the call to `prism_terminate`; in this case, `MPI_Finalize` must be called somewhere after `prism_terminate` in order to shut down the parallel application in a well defined way.

Within `prism_init`, it is detected if the coupled model has been started in the `spawn` or `not_spawn` mode (see 3.1). In `spawn` mode, all spawned processes remain in `prism_init` and participate in the launching of further processes until the spawning of all applications is completed.

Below `prism_init` call, the SCC XML information (see 5.4) is transfered from the Driver to the application process `PSMILe` (see 3.1).

4.1.2 prism_init_comp

`prism_init_comp (comp_id, comp_name, ierror)`

Argument	Intent	Type	Definition
`comp_id`	`Out`	`Integer`	returned component ID
`comp_name`	`In`	`character(len=*)`	name of component in SCC XML file
`ierror`	`Out`	`Integer`	returned error code

Table 4.2: prism_init_comp arguments

`prism_init_comp` needs to be called initially by each process once for each component model executed by the process, no matter if different component models are executed sequentially by the process or if the process is devoted to only one single component model.

If `prism_init` has not been called before by the process, `prism_init_comp` calls it and returns with a warning. Although recommended, it is therefore not necessary to implement a call to `prism_init`.

Below the prism_init_comp call, the component SMIOC XML information (see 5.5) is transfered from the Driver to the component process PSMILe or is read directly by the PSMILe itself in the stand-alone case (see 3.1).

4.1.3 prism_get_localcomn

```
prism_get_localcomm (comp_id, local_comm, ierror)
```

Argument	Intent	Type	Definition
comp_id	In	Integer	component ID or PRISM_Appl_id
local_comm	Out	Integer	returned MPI communicator to be used by the component or the application for its internal communication
ierror	Out	Integer	returned error code

Table 4.3: prism_get_localcomm arguments

MPI communicators for the application and the component model internal communication, separated from the MPI communicators used for coupling exchanges, are provided by the PSMILe and can be accessed via prism_get_local_comm.

If comp_id argument is the component ID returned by routine prism_init_comp, local_comm is a communicator gathering all component processes which called prism_init_comp with the same comp_name argument; if instead, the predefined named integer PRISM_appl_id is provided, the returned local_comm is a communicator gathering all processes of the application.

This routine needs to be called only by MPI parallel code; it is the only MPI specific call in the PSMILe API.

4.1.4 prism_initialized

```
prism_initialized (flag, ierror)
```

Argument	Intent	Type	Definition
flag	Out	Logical	logical indicating whether prism_init was already called or not
ierror	Out	Integer	returned error code

Table 4.4: prism_initialized arguments

This routine checks if prism_init has been called before. If flag is true, prism_init was successfully called; if flag is false, prism_init was not called yet.

4.2 Retrieval of SCC XML information

This section presents PSMILe routine that can be used in the application code to retrieve SCC XML information (see 5.4).

4.2.1 prism_get_nb_ranklists

prism_get_nb_ranklists (comp_name, nb_ranklists, ierror)

Argument	Intent	Type	Definition
comp_name	In	character(len=*)	name of the component in the SCC XML file
nb_ranklists	Out	Integer	number of rank lists for the component in the SCC file
ierror	Out	Integer	returned error code

Table 4.5: prism_get_nb_ranklists arguments

This routine needs to be called before prism_get_ranklists (see 4.2.2) to obtain the number of rank lists that are specified for the component model in the SCC XML file (i.e. the number of elements rank specified for the element component, see 5.4).

4.2.2 prism_get_ranklists

prism_get_ranklists (comp_name, nb_ranklists,ranklists, ierror)

Argument	Intent	Type	Definition
comp_name	In	character(len=*)	name of the component in the SCC XML file
nb_ranklists	In	Integer	number of rank lists
ranklists	Out	Integer	Array(nb_ranklists,3) containing for the nb_ranklists lists of component ranks: a minimum value (nb_ranklists,1), a maximum value (nb_ranklists,2), an increment value (nb_ranklists,3).
ierror	Out	Integer	returned error code

Table 4.6: prism_get_ranklists arguments

This routine returns the lists of ranks that are specified for the component in the SCC XML file. The ranks are the numbers of the application processes used to run the component model; in the SCC XML file, the component model ranks are given as lists of 3 numbers giving, in each list, a minimum value, a maximum value, and an increment value (see also section 5.4). For example, if processes numbered 0 to 7 are used to run a component model, this can be describe with one rank list (0, 7, 1); if processes 0 to 2 and 5 to 7 are used, this can be described with two rank lists (0, 2, 1) and (5, 7, 1). If no maximum values is specified in the SCC file the maximum value is set to the minimum value. If no increment is specified the increment is set to 1.

Rationale: The application rank lists may be needed before the call to prism_init_comp in order to run the components according to the rank lists. Since a component ID is available only after the call to prism_init_comp, the component name is required as input argument to the prism_get_ranklists call instead of the component ID.

4.3 Grids and related quantities definition

In order to describe the grids on which the variables of component models are placed, the following approach was chosen.

The first step is to declare a grid (see `prism_def_grid` in 4.3.1). The grid volume elements which discretize the sphere need then to be defined by providing the corner points (vertices) of these volume elements (see `prism_set_corners` in 4.3.2). At this time, other properties of these volume elements can also provided, such as the volume element mask (see `prism_set_mask` in 4.3.3).

In a second step, different sets of points on which the component model calculates its variables can be placed in these volume elements. Usually, there will be only one definition of volume elements per grid but a larger number of sets of points for different variables on the same grid. The model developer describes where the points are located (see `prism_set_points` in 4.3.6). Points can represent means, extrema or other properties of the variables within the volume.

3D description of all grids

All grids have to be described as covering a 3D domain. A 2D surface in a 3D space necessarily requires information about the location in the third dimension. For example, the grid used in an ocean model to calculate the field of sea surface temperature (SST) would be described vertically by a coordinate array of extent 1 in the vertical direction; the (only) level at which the SST field is calculated would be defined (`prism_set_points`) as well as its vertical bounds (`prism_set_corners`).

Fields not located on a geographical grid ('gridless' grids)

The description of the grid and related quantities is done locally for the domain treated by the local process. The communication patterns used to exchange the coupling fields will usually be based on the geographical description of the local process domain. **Note that the IO of fields located on a non-geographical grid are not supported in the current OASIS4 version**. For fields located on a non-geographical grid, the coupling exchanges are also supported, based on the description of the local process partition in terms of indices in the global index space (see 4.3.1 and 4.3.4).

4.3.1 prism_def_grid

```
prism_def_grid (grid_id, grid_name, comp_id, grid_valid_shape, grid_type,
                ierror)
```

Argument	Intent	Type	Definition
grid_id	Out	Integer	returned grid ID
grid_name	In	character(len=*)	name of the grid (see below)
comp_id	In	Integer	component ID as provided by prism_init_comp
grid_valid_shape	In	Integer	array(2, ndim) (see Table 4.8) giving for each dimension the minimum and maximum index of the valid range (see below)
grid_type	In	Integer	PRISM integer named parameter describing the grid structure (see Table 4.8)
ierror	Out	Integer	returned error code

Table 4.7: prism_def_grid arguments

This routine declares a grid and describes its structure.

- grid_name

  The argument grid_name must match the attribute 'local_name' of the corresponding element 'grid' in the PMIOD and SMIOC XML files and must be unique within the component.

- grid_valid_shape

  The array grid_valid_shape is dimensioned (2,ndim) and gives, for each of the ndim dimensions (see Table 4.8), the minimum and maximum local index values corresponding to the "valid" part of the arrays treated by the process, without the halo region, i.e. $iloc_{low}, iloc_{high}, jloc_{low}, jloc_{high}$ on figure 4.1. For example, if the extent of the first dimension is 100, it may be that the "valid" part of the array goes from 2 to 98.

- grid_type

  The argument grid_type describes the grid type and implicitly specifies the shape of the grid and field arrays passed to the PSMILe. Grids that are currently supported cover:

  – in the horizontal: regular, irregular, Gaussian reduced (and unstructured for I/O only)

  – in the vertical: regular (and unstructured for I/O only)

  Non-geographical grids ('gridless' grids) are also supported for repartitioning, **but not for I/O** (in the current version).

  Table 4.8 lists:

  – the possible values of grid_type for the different grids supported by PSMILe;

  – the corresponding shape of the grid arrays points_1st_array, points_2nd_array, points_3rd_array in prism_set_points;

  – the corresponding shape of the arrays mask_array, and var_array respectively in prism_set_mask, and prism_put / prism_get ;

  – corresponding number of dimensions ndim.

  Other characteristics of the grid will be described by other routines, and the link will be made by the grid identifier grid_id.

Gaussian reduced grids. For Gaussian reduced grids, all processes defining the grid have to call prism_def_grid with grid_type=PRISM_gaussreduced_regvrt. Two numerical dimensions

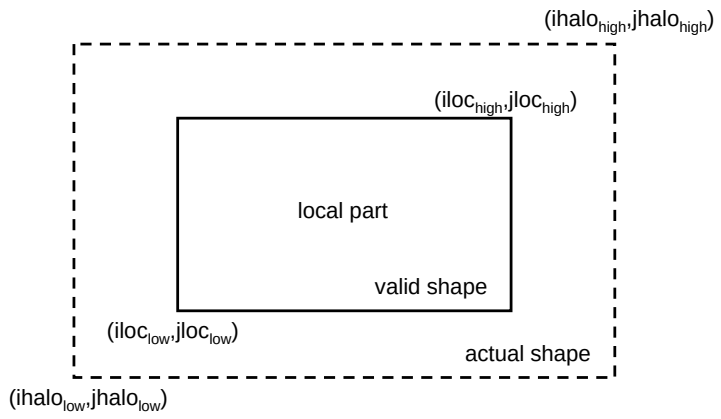


Figure 4.1: Valid shape and actual shape

grid_type	1st_array	2nd_array	3rd_array	mask_array var_array	ndim
PRISM_gridless					3
PRISM_reglonlatvrt	(i)	(j)	(k)	(i,j,k)	3
PRISM_gaussreduced_regvrt	(npt_hor)	(npt_hor)	(k)	(npt_hor,k)	2
PRISM_irrlonlat_regvrt	(i,j)	(i,j)	(k)	(i,j,k)	3
PRISM_unstructlonlat_regvrt	*(npt_hor)*	*(npt_hor)*	*z(k)*	*(npt_hor,k)*	*2*
PRISM_unstructlonlatvrt	*(npt_tot)*	*(npt_tot)*	*(npt_tot)*	*(npt_tot)*	*1*

Table 4.8: Possible values of `grid_type` for the different grids supported by `PSMILe`. `PRISM_unstructlonlat_regvrt` and `PRISM_unstructlonlatvrt` are supported for I/O only.`PRISM_gridless` is supported for repartitioning only.

(`ndim`=2) are used to describe the 3D domain: the first dimension covers the horizontal plane and the second dimension covers the vertical. Furthermore, all these processes have to provide a description of the global reduced gaussian grid by a call to `prism_reducedgrid_map` (see 4.3.5), and have to describe the local partition of the grid with a call to `prism_def_partition` (unless there is no partitioning or if the partitioning is only vertically i.e. level per level, see 4.3.4).

Non-geographical grids. For fields located on a non-geographical grid, `prism_def_grid` still has to be called with `grid_type = PRISM_gridless`. For coding reasons, `ndim` must be always equal to 3 in this case; if in fact the 2nd and/or 3rd dimensions do not exist, the call to `prism_def_grid` must be done with `grid_valid_shape(1:2, 2)` and/or `grid_valid_shape(1:2, 3)` equal to `PRISM_undefined`. The partitioning of non-geographical grids must also be described by a call to `prism_def_partition` (see 4.3.4); furthermore, a call to `prism_set_points_gridless` (see 4.3.7) is also required.

Unstructured grids. The `PSMILe` API as it is currently defined is able to receive and store coordinates of unstuctured grids (see grid types `PRISM_unstructlonlat_regvrt`, `PRISM_unstructlonlatvrt`). I/O of fields defined on unstructured grids are supported as long as the partition definition of the grid is provided with a call to `prism_def_partition` (see 4.3.4). However, as additional information and

related API routines would have to be defined for processing coupling fields provided on those grids, coupling of fields defined on an unstructured grid is not covered yet.

4.3.2 prism_set_corners

```
prism_set_corners (grid_id, nc, corner_actual_shape, corner_1st_array,
                   corner_2nd_array, corner_3rd_array, ierror)
```

Argument	Intent	Type	Definition
grid_id	In	Integer	grid ID returned by prism_def_grid
nc	In	Integer	total number of corners for each volume element
corner_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of corner_xxx_array the minimum and maximum index of the actual range (see below)
corner_1st_array	In	Real or Double	corner longitude (see Table 4.10)
corner_2nd_array	In	Real or Double	corner latitude (see Table 4.10)
corner_3rd_array	In	Real or Double	corner vertical position (see Table 4.10)
ierror	Out	Integer	returned error code

Table 4.9: prism_set_corners arguments

For geographical grids, the volume elements which discretize the computing domain covered locally by the process are defined by giving the corner points (vertices) of those volume elements. The exchange and repartitioning between two coupled component models of a field provided on a geographical grid will be based on this geographical description of the local partition.

- corner_actual_shape

  The array corner_actual_shape is dimensioned (2,ndim) and gives, for each of the ndim dimensions (see Table 4.8), the minimum and maximum local index values corresponding to the "actual" part of the arrays treated by the process including halo regions. corner_actual_shape is therefore greater or equal to the grid_valid_shape (see section 4.3.1). For example, if the actual extent of the first dimension is 100, it may be that the valid index goes from 0 to 99, or from 1 to 100.

- corner_xxx_array

  Units of corner_xxx_array

  Units of arrays corner_xxx_array describing the grid volume elements should be indicated in the PMIOD and SMIOC XML files; they are not included in the prism_set_corners call.

  Currently, the array corner_1st_array must be provided in degrees East; there is no particular restriction in the numbers used (e.g. numbers greater than 360, or negative numbers are supported) but longitudes of the corners of one cell have to define the size of the cell (e.g. a cell with corners at 5 and 355 is a cell of 350 degrees, not a cell of 10 degrees). Currently, the array corner_2nd_array must be provided in degrees North (spherical coordinate system); negative numbers are of course supported. Other units will eventually be supported later when appropriate automatic conversion will be implemented.

  For corner_3rd_array, units must be the same on the source and target sides.

  Shape of corner_actual_shape

  Table 4.10 gives the expected shape of the corner_xxx_array for the various grid_type. When applicable, Fortran ordering must be used to define the corners. For PRISM_irrlonlat_regvrt, the corners have to be given counterclockwise looking in the k positive direction.

grid_type	corner_1st_array	corner_2nd_array	corner_3rd_array
PRISM_reglonlatvrt	(i,2)	(j,2)	(k,2)
PRISM_gaussreduced_regvrt	(npt_hor,2)	(npt_hor,2)	(k,2)
PRISM_irrlonlat_regvrt	(i,j, nc_{half})	(i,j, nc_{half})	(k,2)

Table 4.10: Dimensions of corner_xxx_arrays for the various grid_type; nc is the total number of corners for each volume element; nc_{half} is nc divided by 2; i, j, k, npt_hor are the extent of the respective numerical dimensions (see Table 4.8).

4.3.3 prism_set_mask

```
prism_set_mask(mask_id, grid_id, mask_actual_shape, mask_array,
               new_mask, ierror)
```

Argument	Intent	Type	Definition
mask_id	InOut	Integer	mask ID
grid_id	In	Integer	grid ID returned by prism_def_grid
mask_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of mask_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
mask_array	In	Logical	array of logicals; see Table 4.8 for its dimensions; if an array element is .true. (.false.), the corresponding field grid point is (is not) valid.
new_mask	In	Logical	if .true. a mask is specified for the first time for this mask_id (Out); if .false. mask values for this mask_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.11: prism_set_mask arguments

This routine defines a mask array. Different masks can be defined for the same grid. One particular mask will be attached to a field by specifying the corresponding mask_id in the prism_def_var call used to declare the field (see section 4.4.1).

4.3.4 prism_def_partition

```
prism_def_partition (grid_id, nbr_subdomains, offset_array, extent_array,
                     ierror)
```

The local partition treated by the model process can also be described in term of indices in the global index space with a call to prism_def_partition . Calling this routine is mandatory for the grids listed below.

The global index space is a unique and common indexing for all grid points of the component model. For example, if a component model covers a global domain of 200 grid points that is distributed over two processes covering 100 points each, the first and second partition **local** indices can both be (1:100); however, their **global** indices will be respectively (1:100) and (101:200).

A partition may also cover different sets of points disconnected in the global index space; each one of those sets of point constitutes one subdomain and has to be described by its offset and extent in the global index space. Let's suppose, for example, that the 200 grid points of a component model are distributed over two processes such that points 1 to 50 and 76 to 100 are treated by the first process and

Argument	Intent	Type	Definition
grid_id	In	Integer	grid ID returned by prism_def_grid
nbr_subdomains	In	Integer	number of subdomains, in the global index space, covered by the grid_valid_shape domain
offset_array	In	Integer	array(nbr_subdomains, ndim) containing for each subdomain the offset in each ndim dimension in the global index space.
extent_array	In	Integer	array(nbr_subdomains, ndim) containing for each subdomain the extent in each ndim dimension in the global index space.
ierror	Out	Integer	returned error code

Table 4.12: prism_def_partition arguments

such that points 51 to 75 and 101 to 200 are treated by the second process. In this case, the number of subdomains for each process is 2, and the first process subdomains can be described with global offsets of 0 and 75 (offset_array(1,1)=0, offset_array(2,1)=75) and extents of 50 and 25 (extent_array(1,1)=50, extent_array(2,1)=25), while the second process subdomains can be described by global offsets of 50 and 100 (offset_array(1,1)=50, offset_array(2,1)=100) and extent of 25 and 100 (extent_array(1,1)=25, extent_array(2,1)=100)[1].

Gaussian reduced grids . For those grids, prism_def_partition must be called by each model process to describe its local partition, unless there is no partitioning or if the partitioning is done only vertically (i.e. level per level). In this OASIS4 version, the horizontal partitioning, if any, must be the same for all vertical levels; therefore, offset_array(:,2) must always be equal 0 and extent_array(:,2) must always be equal to the number of vertical levels. The horizontal partitioning, described by offset_array(:,1) and extent_array(:,1), must describe each latitude band of the reduced grid local partition as a subdomain on its own. The offset_array(:,1) refer to the offset of each subdomain in a horizontal global index space defined as the sequence of points starting at the most northern (or southern) latitude band and is going down in circular manner to the most southern (or northern) latitude band. Note that in addition all processes have to call prism_reducedgrid_map for a description of the global reduced Gaussian grid (see 4.3.5).

Unstructured grids. For I/O of a field given on an unstructured grid, a call to prism_def_partition is mandatory to use the parallel I/O mode (see section 4.6).

Non-geographical grids ('gridless' grids). Coupling exchanges **(but not I/O in the current version)** of fields not located on a geographical grid are supported, based on the description of the process local partition in terms of indices in the global index space. A call to prism_def_partition is therefore mandatory for such grids.

4.3.5 prism_reducedgrid_map

```
prism_reducedgrid_map (grid_id, nbr_latitudes, nbr_points_per_lat, ierror)
```

Argument	Intent	Type	Definition
grid_id	In	Integer	grid ID returned by prism_def_grid
nbr_latitudes	In	Integer	number of latitudes of the global grid
nbr_points_per_lat	In	Integer	array(nbr_latitudes) containing for each latitude the number of grid points in longitude.
ierror	Out	Integer	returned error code

Table 4.13: prism_reducedgrid_map arguments

[1] Note that this example supposes ndim=1.

For Gaussian reduced grids only. All processes that announce a Gaussian reduced grid have to call `prism_reducedgrid_map` for a description of the global reduced Gaussian grid, providing the same identical information about the global grid..

4.3.6 prism_set_points

```
prism_set_points ( point_id, point_name, grid_id, points_actual_shape,
                   points_1st_array, points_2nd_array, points_3rd_array,
                   new_points, ierror)
```

Argument	Intent	Type	Definition
point_id	InOut	Integer	ID for the set of points
point_name	In	character(len=*)	name of the set of points: must match the attribute 'local_name' of the corresponding element 'points' in the PMIOD and SMIOC XML files and must be unique within the component.
grid_id	In	Integer	grid ID returned by prism_def_grid
points_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of points_xxx_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
points_1st_array	In	Real or Double	array giving the longitudes for this set of grid points; see Table 4.8 for its dimensions
points_2nd_array	In	Real or Double	array giving the latitudes for this set of grid points; see Table 4.8 for its dimensions
points_3rd_array	In	Real or Double	array giving the vertical positions for this set the grid points; see Table 4.8 for its dimensions
new_points	In	Logical	if .true. points are specified for the first time for this point_id (Out); if .false. points for this point_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.14: prism_set_points arguments

With prism_set_points the model developer describes the geographical location of the variables on the grid. Variables can represent means, extrema or other properties of the variables within volume. Different sets of points can be defined for the same grid (staggered grids); each set will have a different point_id. A full 3D description has to be provided; for example, a set of points discretizing a 2D surface must be given a vertical position. Units for points_1st_array, points_2nd_array and points_3rd_array must be respectively the same than the ones for corner_1st_array, corner_2nd_array and corner_3rd_array (see section 4.3.2).

Non-geographical grids. For non-geographical grids ('gridless' grids), prism_set_points_gridless should be called instead of prism_set_points (see 4.3.7).

4.3.7 prism_set_points_gridless

```
prism_set_points_gridless( point_id, point_name, grid_id, new_points, ierror)
```

Argument	Intent	Type	Definition
point_id	InOut	Integer	set of points ID
point_name	In	character(len=*)	name of the set of points: must match the attribute 'local_name' of the corresponding element 'points' in the PMIOD and SMIOC XML files and must be unique within the component.
grid_id	In	Integer	grid ID returned by prism_def_grid
new_points	In	Logical	if .true. points are specified for the first time for this point_id (Out); if .false. points for this point_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.15: prism_set_points_gridless arguments

The routine prism_set_points_gridless has to be called for non-geographical grids to retrieve a grid point ID.

4.3.8 prism_set_vector

```
prism_set_vector (vector_id, vector_name, array_ids, new_vector, ierror)
```

Argument	Intent	Type	Definition
vector_id	InOut	Integer	ID of the vector sets of points
vector_name	In	character(len=*)	name of the vector set of points: must match the attribute 'local_name' of the corresponding element 'vector' in the PMIOD and SMIOC XML files and must be unique within the component.
array_ids	In	Integer	array(3) containing the point_ids returned by previous calls to prism_set_points used to define the set of points for each vector component
new_vector	In	Logical	if .true. vector sets of points are specified for the first time for this vector_id (Out); if .false. vector sets of points for this vector_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.16: prism_set_vector arguments

For vector fields, sets of points which have been defined for each vector component by a previous call to prism_set_points can be linked together with a call to prism_set_vector (e.g. on a Arakawa C grid all three vector components are located on different sets of point in the physical space). In any case, three valid point_ids need to be specified in array_ids. *I/O of vector fields are currently supported but coupling of vector fields are not.*

4.4 Declaration of Coupling/IO fields

4.4.1 prism_def_var

```
prism_def_var(var_id, var_name, grid_id, point_id, mask_id, var_nodims,
              var_actual_shape, var_type, ierror)
```

Argument	Intent	Type	Definition
var_id	Out	Integer	returned field ID
var_name	In	character(len=*)	name of the field: must correspond to the attribute local_name of element transient in the PMIOD and SMIOX XML files and must be unique within the component
grid_id	In	Integer	ID of the field grid (as returned by prism_def_grid)
point_id	In	Integer	ID of the field set of points as returned by prism_set_points(for scalar field), or ID of the vector sets of points as returned by prism_set_vector(for vector field)
mask_id	In	Integer	ID of the field mask as returned by prism_set_mask, or ID of the set of 3 masks as returned by prism_set_vectormask (for vector field), or PRISM_UNDEFINED.
var_nodims	In	Integer	var_nodims(1): the number of dimensions of var_array that will contain the coupling/IO field (see 4.6), i.e. ndim (see Table 4.8) except for vector for which it is ndim+1; var_nodims(2): number of vector components (3), 0 otherwise.
var_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of var_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
var_type	In	Integer	field type: PRISM integer named parameter PRISM_Integer, PRISM_Real or PRISM_Double_Precision
ierror	Out	Integer	returned error code

Table 4.17: prism_def_var arguments

After the initialisation and grid definition phases, each field that will be send/received to/from another component model (coupling field) or that will be written/read to/from a disk file (IO field) through PSMILe'put'/'send' actions needs to be declared and associated with a previously defined grid and mask.

The units of a coupling/IO field should be indicated in the PMIOD XML file, not in its declaration call. By consulting the appropriate PMIOD, the user is able to check if the units of a coupling field match on the source and target side and if not, he has to choose appropriate transformations in the SMIOCs.

4.5 Neighborhood search and determination of communication patterns

4.5.1 prism_enddef

```
prism_enddef (ierror)
```

Argument	Intent	Type	Definition
ierror	Out	Integer	returned error code

Table 4.18: prism_enddef arguments

Following prism_init, prism_enddef is the second collective call and has to be called once by each application process when all components within the application have completed their definition phase.

To perform the exchange of coupling fields during the run, it is required to establish communication only between those pairs of processes that actually have to exchange data based on the user defined coupling configuration in the SMIOCs XML files (see section 5.5).

For each coupling exchange involving a regridding between the source and the target grids, the neighborhood search is performed. It identifies, for each grid point of each target process, the source grid points and corresponding source process that will be used to calculate the target grid point value. For a coupling exchange involving only repartitioning, each target grid point corresponds exactly to only one source grid point; in this case the 'neighborhood search' process identifies, for each grid point of each target process, on which source process the matching source grid point is located.

In order to save memory and CPU time in the neighbourhood search and the establishment of the communication patterns, prism_enddef works in a parallel way on the local grid domain covered by each application process as much as possible. In an initial step, each process calculates a bounding box covering its local geographical volume domain previously defined by prism_set_corners (see section 4.3.2). The bouding boxes of all processes are sent to and collected by all processes. Each source process calculates the intersection of its bounding box with each other process bounding box, thereby identifying the potential interpolation partners and corresponding bounding box intersection. (For fields located on non-geographical fields, see 4.3.1, the intersection calculation is based on the local domain description in the global index space, see 4.3.4.) For each bounding box intersection, the source process creates a sequence of simplified grids and corresponding bounding boxes, each one coarsened by a factor of 2 with respect to the previous one, until falling back onto the bounding box covering the whole intersection (similar to a Multigrid Algorithm). Starting on the coarsest level the search algorithm determines, at each multigrid level, the source bounding box for each target grid point in the intersection. When the bounding box at the finer level is identified, the neighbours of the target grid point, i.e. the source points participating in its calculation (regridding case) or the matching source grid point (repartitioning only case), are identified. For each intersection of source and target grid processes, the 'Ensemble of grid Points participating in the Interpolation Operation (EPIO)' (or in the repartitioning) on the source side (EPIOS) and on the target side (EPIOT) are identified. The results of this search are transfered to the target process. For the coupling exchange involving regridding, the EPIOS and EPIOT definition and all related grid information are also transferred to the Transformer (see section 3.2).

As the results of the neighbourhood search are known in the source PSMILe, only the usefull grid points will be effectively sent later on during the coupling exchanges, minimizing the amount of data to be transferred.

4.6 Exchange of coupling and I/O fields

The PSMILe exchanges are based on the principle of "end-point" data exchange. When producing data, no assumptions are made in the source component code concerning which other component will consume these data or whether they will be written to a file, and at which frequency. Likewise, when asking for data a target component does not know which other component model produces them or whether they are read in from a file. The target or the source (another component model or a file) for each field is defined by the user in the SMIOC XML file (see section 5.5) and the coupling exchanges and/or the I/O actions take place according to the user external specifications. The switch between the coupled mode and the forced mode is therefore totally transparent for the component model. Furthermore, source data can be directed to more than one target (other component models and/or disk files).

The sending and receiving PSMILe calls prism_put and prism_get can be placed anywhere in the source and target code and possibly at different locations for the different coupling fields. These routines can be called by the model at each timestep. The actual date at which the call is performed and the date bounds for which it is valid are given as arguments; the sending/receiving is actually performed only if the date and date bounds corresponds to a time at which it should be activated, given the field coupling or I/O dates indicated by the user in the SMIOC; a change in the coupling or I/O dates is therefore also totally transparent for the component model itself. The PSMILe can also take into account a timelag between the sending prism_put and the corresponding prism_get defined by the user in the SMIOC (see item 6. of section 5.5.4).

Local transformations can be performed in the source component PSMILe below the prism_put and/or in the target component PSMILe below the prism_get like time accumulation, time averaging, algebraic operations, statistics, scattering, gathering (see item 7. of section 5.5.4 and item 5. of section 5.5.5).

When the action is activated at a coupling or I/O date, each process sends or receives only its local partition of the data, corresponding to its local grid defined previously. The coupling exchange, including data repartitioning if needed, then occurs, either directly between the component models, or via additional Transformer processes if regridding needed (see section 3.2).

If the user specifies that the source of a prism_get or the target of a prism_put is a disk file, the PSMILe exploits the GFDL mpp_io package (2) for its file I/O. The supported file format is NetCDF according to the CF convention (4). The mpp_io package is driven by a PSMILe internal layer which interfaces with various sources of information. For instance, the definition of grids and masks as well as the form of the data (bundle or vector) of a field is provided through the PSMILe API. On the other hand the information with regard to the CF standard name and unit are provided by the SMIOC XML file through the Driver.

The mpp_io package can operate in three general I/O modes:

- **Distributed I/O**

  Each process works on a individual file containing the I/O field on the domain onto which that process works. Domain partitioning information is written into the resulting files such they can be merged into one file during a post processing step.

- **Pseudo parallel I/O**

  The whole field is read from or written to one file. The domain partitioning information is exploited such that the data are collected - stitched together - during the write operation or distributed to the parallel processes of a component model during the read operation. This domain stitching or distribution is automatically done by the PSMILe on the component model master process and happens transparently for the parallel component model itself. For unstructured grids, this mode is supported only if the definion of the local partition in terms of indices in the global index space is provided with prism_def_partition (see section 4.3.4).

- **Parallel I/O**

  A fully parallel I/O using the parallel NetCDF (7) library and MPI-IO is available. This allows

parallel IO of distributed data into a single NetCDF file which is controlled by MPI-IO instead of collecting the data on the master process first. To have this feature available the PSMILe has to be linked against the parallel NetCDF library. The PSMILe library has to be generated with `-D__PARNETCDF`. Note that this type of IO is not yet supported for applications having more than 1 component.

The `PSMILe` I/O layer also copes with the fact that the input data may be spread accross a number of different files[2], and that NetCDF file format has certain restrictions with respect to size of a file. Therefore, on output chunking of a series of time stamps across multiple files will be provided depending on a threshold value of the file size.

4.6.1 prism_put

`prism_put (var_id, date, date_bounds, var_array, info, ierror)`

Argument	Intent	Type	Definition
`var_id`	`In`	`Integer`	field ID returned from `prism_def_var`
`date`	`In`	`Type(PRISM_Time_Struct)`	date at which the `prism_put` is performed
`date_bounds`	`In`	`Type(PRISM_Time_Struct)`	array(2) giving the date bounds between which this call is valid
`var_array`	`In`	`Integer, Real or Double`	field array to be sent (see Table 4.8 for its dimensions)
`info`	`Out`	`Integer`	returned info about action performed (see below)
`ierror`	`Out`	`Integer`	returned error code

Table 4.19: prism_put arguments

This routine should be called to send `var_array` content to a target component or file. The target is defined by the user in the SMIOC XML files (see section 5.5). This routine can be called in the component model code at each timestep; the actual date at which the call is performed and the date bounds for which it is valid must be given as arguments as `PRISM_Time_Struct` structures (see `/prism/src/lib/common_oa4/src/prism_constants.F90`) ; the sending is actually performed only if the date and date bounds corresponds to a time at which it should be activated, given the field coupling or I/O dates indicated by the user in the SMIOC XML file.

The meaning of the different `info` returned are as follows:

- PRISM_NoAction = 0: no action is performed for this call
- PRISM_Cpl = 1000: the array is only sent to another component
- PRISM_CplIO = 1100: the array is sent to another component and written to a file
- PRISM_CplRst = 1010: the array is sent to another component and written to a coupling restart file
- PRISM_CplTimeop = 1001: the array is sent to another component and used in a time operation (accumulation, averaging)
- PRISM_CplIORst = 1110: the array is sent to another component, written to a file, and written to a coupling restart file

[2]The system calls 'scandir' and 'alphasort' are used to implement this feature (see routine `/prism/src/lib/psmile_oa4/src_scandir.c`). In case of problems with these system calls, one may try to compile with the -D__MYALPHASORT. If there are still problems, one has to comment the calls to `psmile_io_scandir_no_of_files` and `psmile_io_scandir` in `psmile_open_file_byid.F90`, but then that `PSMILe` functionality will not be provided anymore.

- PRISM_CplIOTimeop = 1101: the array is sent to another component, written to a file, and used in a time operation
- PRISM_CplRstTimeop = 1011: the array is sent to another component, written to a coupling restart file, and used in a time operation
- PRISM_CplIORstTimeop = 1111: the array is sent to another component, written to a file, written to a coupling restart file, and used in a time operation
- PRISM_IO = 100: the array is only written to a file
- PRISM_IORst = 110: the array is written to a file and to a coupling restart file
- PRISM_IOTimeop = 101: the array is written to a file and used in a time operation
- PRISM_IORstTimeop = 111: the array is written to a file and to a coupling restart file and is used in a time operation
- PRISM_Rst = 10: the array is only written to a coupling restart file
- PRISM_RstTimeop = 11: the array is written to a coupling restart file and used in a time operation
- PRISM_Timeop = 1: the array is used in a time operation

The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 4.8.3).

This routine will return even if the corresponding `prism_get` has not been performed on the target side, both for an exchange through the Transformer and for a direct exchange (as the content of the `var_array` is buffered in the `PSMILe`).

4.6.2 prism_get

```
prism_get(var_id, date, date_bounds, var_array, info, ierror)
```

Argument	Intent	Type	Definition
`var_id`	`In`	`Integer`	field ID returned by `prism_def_var`
`date`	`In`	`Type(PRISM_Time_Struct)`	date at which the `prism_get` is performed
`date_bounds`	`In`	`Type(PRISM_Time_Struct)`	array(2) giving the date bounds between which this call is valid
`var_array`	`InOut`	`Integer, Real or Double`	field array to be received (see Table 4.8 for its dimensions)
`info`	`Out`	`Integer`	returned info about action performed (see below)
`ierror`	`Out`	`Integer`	returned error code

Table 4.20: prism_get arguments

This routine should be called to receive a field `var_array` from a source component or file. The source is defined by the user in the SMIOC XML files (see section 5.5). As for `prism_put` , this routine can be called in the component model code at each timestep; the actual date at which the call is performed and the date bounds for which it is valid must be given as arguments; the receiving is actually performed only if the date and date bounds corresponds to a time at which it should be activated, given the field coupling or I/O dates indicated by the user in the SMIOC XML file.

Note that `var_array` is of intent `InOut`. It is therefore updated only for the part for which data have been effectively received. We therefore recommend to initialize `var_array` with `PRISM_Undefined` (=-65535, see `prism/src/lib/common_oa4/include/prism.inc`) before the `prism_get` to be able to clearly identify the data received.

The meaning of the different `info` returned are as follows:

- PRISM_NoAction = 0: no action is performed for this call
- PRISM_Cpl = 1000: the array is only received from another component
- PRISM_IO = 100: the array is read from a file
- PRISM_IOTimeop = 101: the array is read from a file and used in a time operation

The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 4.8.3).

This routine will return only when the corresponding `prism_put` is performed on the source side and when data is available in `var_array`, after regridding if needed.

4.6.3 prism_put_inquire

```
prism_put_inquire (var_id, date, date_bounds, info, ierror)
```

Argument	Intent	Type	Definition
`var_id`	`In`	`Integer`	field ID returned from `prism_def_var`
`date`	`In`	`Type(PRISM_Time_Struct)`	date at which the `prism_put` would be performed
`date_bounds`	`In`	`Type(PRISM_Time_Struct)`	array(2) giving the date bounds between which the field would be valid
`info`	`Out`	`Integer`	returned info about action that would be performed (see below)
`ierror`	`Out`	`Integer`	returned error code

Table 4.21: prism_put_inquire arguments

This function is called to inquire if the corresponding `prism_put` (i.e. for same `var_id`, `date`, and `date_bounds`) would effectively be activated. This can be useful if the calculation of the related `var_array` is CPU consuming.

The meaning of the different `info` returned are as for the `prism_put` routine (see 4.6.1).

The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 4.8.3).

4.6.4 prism_put_restart

```
prism_put_restart (var_id, date, date_bounds, data_array, info, ierror)
```

This function forces the writing of a field into a coupling restart file.

If a coupling restart file of a coupling field is needed[3] but not available, it might be useful to run the source component model beforehand to create the first coupling restart file of an experiment explicitly with a call to `prism_put_restart` . The returned `info` should always be PRISM_Rst = 10 (the array is only written to a coupling restart file). The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 4.8.3).

To use `prism_put_restart` , one should pay attention to the following details:

- There must be a lag equal to 0 defined for the field in the appropriate smioc XML file (see 5.5.4) .

[3]For coupling fields with lag > 0 (see element `lag` in section 5.5.4), a coupling restart file is needed to start the run. In this case, two restart files are opened, one for reading and one for writing. At the beginning of a run, the respective source `PSMILe` processes read their local part of the coupling field in the coupling restart file during the `prism_enddef` phase and send it to the Transformer which performs the interpolation and sends the interpolated field to the target component model. Below the last call to `prism_put` in the run, the coupling field is also automatically written to its writing coupling restart file; in this case the <date> is the current run end date.

Argument	Intent	Type	Definition
var_id	In	Integer	transient handle from prism_def_var
date	In	Type(PRISM_Time_Struct)	date at which the prism_put_restart is performed
date_bounds	In	Type(PRISM_Time_Struct)	array dimensioned (2) giving the date bounds between which this data is valid
data_array	In	Integer, Real or Double	data array to be transferred
info	Out	Integer	returned info about action performed
ierror	Out	Integer	returned error code

Table 4.22: prism_put_restart arguments

- Since the prism_enddef performs some IO related initialisation, a prism_put_restart cannot be invoked before the prism_enddef is completed.
- The time information for each data set that is written into the restart file corresponds to the upper boundary of the time interval which is represented by the data set. To restart from a particular data set the job start date indicated in the SCC.XML needs to correspond to the required time info in the restart file.
- Currently it is only possible to dump raw fields into the NetCDF file. Fields written to a restart file via prism_put_restart are currently taken as is and are not processed with respect to local operations like gathering/scattering averaging, summation or any reduction operations.
- The name of the reading restart file will be <field_local_name>_<component_local_name> -_<application_local_name>_rst.<date>, where <date> is the current run start date.

A concrete example on how to use the PSMILe prism_put_restart routine to create an OASIS4 coupling restart file can be found in directory prism/src/mod/oasis4/examples/create_restart (see the README therein).

4.7 Termination Phase

4.7.1 prism_terminate

```
prism_terminate (ierror)
```

Argument	Intent	Type	Definition
ierror	Out	Integer	returned error code

Table 4.23: prism_terminate arguments

In analogy to the initialisation phase, a call to prism_terminate, which again is a collective call, will make the calling process to wait for other processes participating in the coupling to reach the prism_terminate as well. At this point, the following actions are performed:

- All open units under control of the PSMILe are closed.
- The output to standard out is flushed.
- The Driver is notified about the termination of the respective process.
- All memory under control of PSMILe is deallocated.

After calling prism_terminate , no coupling exchanges are possible for this process and no further I/O actions under control of the PSMILe can be performed; however, it is still possible for the application to perform local operations and to write additional output which shall not be under control of the PSMILe. If MPI_Init has been called in the code before the call to prism_init, component internal MPI communication is still possible after the call to prism_terminate, until the MPI_Finalize is called by the component (see also section 4.1.1). Otherwise prism_terminate will call MPI_Finalize.

4.7.2 prism_terminated

```
prism_terminated (flag, ierror)
```

Argument	Intent	Type	Definition
flag	Out	Logical	if .true., prism_terminate was already called
ierror	Out	Integer	returned error code

Table 4.24: prism_terminated arguments

This routine can be used to check whether prism_terminate has already been called by this process. This may help to detect ambiguous implementations of multi-component applications.

4.7.3 prism_abort

```
prism_abort (comp_id, routine, message )
```

Argument	Intent	Type	Definition
comp_id	In	Integer	component ID as provided by prism_init_comp
routine	In	Character	calling routine name
message	In	Character	user defined message

Table 4.25: prism_abort arguments

It is common practice in non parallel Fortran codes to terminate the program by calling a Fortran STOP in case a runtime error is detected. In MPI-parallelized codes it is strongly recommended to call MPI_Abort instead to ensure that all parallel processes are stopped and thus to avoid non-defined termination of the parallel program. For coupled application, the PSMILe provides a prism_abort call which guarantees

a clean and well-defined shut down of the coupled model. We recommend to use `prism_abort` instead of a Fortran `STOP` or a `MPI_Abort`.

4.8 Query and Info Routines

4.8.1 prism_get_calendar_type

`prism_get_calendar_type (calendar_name, calendar_type_id, ierror)`

Argument	Intent	Type	Definition
`calendar_name`	`Out`	`Character(len=132)`	name of calendar used
`calendar_type_id`	`Out`	`Integer`	ID of calendar used
`ierror`	`Out`	`Integer`	returned error code

Table 4.26: prism_get_calendar_type arguments

This routine returns the name and the ID of the calendar used in the `PSMILe`. Currently, the only calendar supported is the 'Proleptic Gregorian Calendar' (i.e. a Gregorian calendar[4] extended to dates before 15 Oct 1582) and its ID is 1 (i.e. the PRISM integer name parameter `PRISM_Cal_Gregorian = 1`, see `prism/src/lib/common_oa4/include/prism.inc`). Simple calendars with 360 and 365 days are implemented but not directly available to the user. In a future version, the calendar type should be chosen and specified by the user in an XML configuration file, read in from this XML file by the Driver, and transfered to the `PSMILe`.

4.8.2 prism_calc_newdate

`prism_calc_newdate (date, date_incr, ierror)`

Argument	Intent	Type	Definition
`date`	`InOut`	`Type(PRISM_Time_Struct)`	In and Out date
`date_incr`	`In`	`Integer, Real or Double`	Increment in seconds to add to the date
`ierror`	`Out`	`Integer`	returned error code

Table 4.27: prism_calc_newdate arguments

This routine adds a time increment of `date_incr` seconds to the `date` given as In argument and returns the result in the `date` as Out argument. The time increment may be negative. For the date structure `PRISM_Time_Struct`, see `/prism/src/lib/common_oa4/src/prism_constants.F90`.

4.8.3 prism_error

`prism_error (ierror, error_message)`

Argument	Intent	Type	Definition
`ierror`	`In`	`Integer`	an error code returned by a `PSMILe` routine
`error_message`	`Out`	`character(len=*)`	corresponding error string

Table 4.28: prism_error arguments

This routine returns the string of the error message `error_message` corresponding to the error code `ierror` returned by other `PSMILe` routines. In general, 0 is returned as error code if the routine completed without error; a positive error code means a severe problem was encountered.

[4]The Gregorian calendar considers a leap year every year which is multiple of 4 but not multiple of 100, and every year which is a multiple of 400.

4.8.4 prism_version

`prism_version()`

This routine prints a message giving the version of the `PSMILe` library currently used.

4.8.5 prism_get_real_kind_type

`prism_get_real_kind_type (kindr, type, ierror)`

Argument	Intent	Type	Definition
`kindr`	`In`	`Integer`	kind type parameter of REAL variables
`type`	`Out`	`Integer`	PRISM datatype corresponding to `kindr`
`ierror`	`Out`	`Integer`	returned error code

Table 4.29: prism_get_real_kind_type arguments

This routine returns in `type` the PRISM datatype which corresponds to the kind type parameter `kindr`. `type` can be either `PRISM_Real = 4`, or `PRISM_Double_Precision = 5` (see `prism/src/lib/common_oa4/include/prism.inc`).

4.8.6 prism_remove_mask

`prism_remove_mask ( mask_id, ierror )`

Argument	Intent	Type	Definition
`mask_id`	`In`	`Integer`	mask ID as returned by `prism_set_mask`
`ierror`	`Out`	`Integer`	returned error code

Table 4.30: prism_remove_mask arguments

The routine removes the mask information linked the mask ID `mask_id` given as argument.

Chapter 5

OASIS4 description and configuration XML files

This chapter details the content of the XML description and specification files used with OASIS4.

- The XML description files are used to:
  - describe each application: the “Application Description” (AD)
  - describe the relations a component model of an application is able to establish with the rest of the coupled model through inputs and outputs: the “Potential Model Input and Output Description” (PMIOD)

  The description XML files, i.e. the ADs and PMIODs, should be created by the component model developer to provide information about the general characteristics and the potential coupling interface of its code, but they are not used by the OASIS4 coupler.

- The XML specification files are used to:
  - specify the general characteristics of a coupled model run: the “Specific Coupling Configuration” (SCC)
  - specify the relations the component model will establish with the rest of the coupled model through inputs and outputs for a specific run: the Specific Model Input and Output Configuration (SMIOC).

  The specification XML files, i.e. the SCC and the SMIOCs, must be created by the coupled model user, i.e. the person that builds the coupled model. They provide specifications about the process management and the coupling and I/O exchanges of one particular coupled model and are used by the OASIS4 coupler.

A Graphical User Interface (GUI) is currently being developed to facilitate the creation of those files. Based on the ADs and PMIODs description files, the GUI will help the user to create the SCC and SMIOCs specification files.

5.1 Introduction to XML concepts

Extensible Markup Language (XML) is a simple, very flexible text format. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. An XML document is simply a file which follows the XML format.

The purpose of a DTD or a Schema is to define the legal building blocks of an XML document. The AD, SCC, PMIOD and SMIOC XML documents must follows the Schemas files `ad.xsd`, `scc.xsd`,

`pmiod.xsd` and `smioc.xsd` respectively, available in the directory `/prism/src/mod/oasis4/util/xmlfiles`.

The `xmllint` command with the following options can be used to validate an XML file `file.xml` against a Schema file `file.xsd`:

```
xmllint --noout --valid --postvalid --schema file.xsd file.xml
```

The building blocks of XML documents are Elements, Tags, and Attributes.

- Elements

  Elements are the main building blocks of XML documents.

  Examples of XML elements in `pmiod.xsd` are `prismcomponent` or `code`. Elements can contain text, other elements, or be empty.

  The values of `minOccurs` and `maxOccurs` for an element in the Schema file indicate how many times this element must occur in the corresponding XML file; if `minOccurs` and `maxOccurs` are not specified, the element must appear once.

- Tags

  Tags are used to markup elements.

  In the XML file, a starting tag like <element_name> mark up the beginning of an element, and an ending tag like </element_name> mark up the end of an element.

  Example: <laboratory>Meteo-France</laboratory>

  An empty element will appear as <element_name />.

- Attributes

  Attributes provide extra information about elements and are placed inside the start tag of an element. As indicated in the Schema file, an attribute may be “required” (use=’required’) or “optional” (use=’optional’).

  Example: <grid local_name=”AT31_2D”>

  The name of the element is “grid”. The name of the attribute is “local_name”. The value of the attribute is “AT31_2D”.

5.2 The Application Description (AD)

The Application Description (AD) describes the general characteristics of one application. There is one AD per application, i.e. per code which when compiled forms one executable. An application may contain one or more component model. This description XML file should be created by the application developer to provide information about the application general characteristics[1] but it is not used by the OASIS4 coupler.

The AD Schema is given in `/prism/src/mod/oasis4/util/xmlfiles/ad.xsd`. The AD file name must be <application_local_name>_ad.xml where <application_local_name> is the application name.

The AD contains the element ‘application’ which is composed of (see the `ad.xsd`):

- the application name: attribute ‘local_name’, which should match argument `appl_name` of `PSMILe` call `prism_init` (see section 4.1.1);
- a description of the application: attribute ‘long_name’;
- the version of the OASIS4 Schema file: attribute ‘oasis4_version’;

[1] On the longer term, in order to avoid duplication of information, it is foreseen to develop a tool to extract automatically all AD information which is already in the code (e.g. the component names given argument `comp_name` of `prism_init_comp` calls).

- the mode into which the application may be started: attribute 'start_mode': 'spawn', 'notspawn' or 'notspawn_or_spawn' (see section 3.1);
- the mode into which the application may run: attribute 'coupling_mode': 'coupled', 'standalone', or 'coupled_or_standalone';
- the arguments with which the application may be launched: element 'argument';
- the total number of processes the application can run on: element 'nbr_procs';
- the platforms on which the application has run: element 'platform';
- the list of components included in the application: element 'component'; for each component:
  - the component name: attribute 'local_name', which should match the argument `comp_name` of `PSMILe` call `prism_init_comp` (see section 4.1.2);
  - a description of the component: attribute 'long_name';
  - the simulated part of the climate system: attribute 'simulated': either `ocean`, `sea_ice`, `ocean_biogeochemistry`, `atmosphere`, `atmospheric_chemistry`, or `land`; if an AD contains more than one component simulating the same part of the climate system, the user will have in the SCC (see below) to choose among these components;
  - whether or not this component is always active in the application: attribute 'default', either `true` or `false`);
  - the number of processes on which the component can run (element 'nbr_procs').

5.3 The Potential Model Input and Output Description (PMIOD)

The Potential Model Input and Output Description (PMIOD) describes the relations a component model is potentially able to establish with the rest of the coupled model through inputs and outputs. There should be one PMIOD per component model, written by the component developer[2] to describe its component potential coupling interface, but the PMIOD files are not used by the OASIS4 coupler.

The PMIOD Schema is given `/prism/src/mod/oasis4/util/xmlfiles/pmiod.xsd`. The PMIOD file name should be <application_local_name>_<component_local_name>_pmiod.xml where <application_local_name> is the application name and <component_local_name> is the component name. Examples of PMIOD xml files for the toy coupled model TOYOA4 can be found in `prism/util/running/toyoa4/input`.

The PMIOD contains 3 types of information:

- general characteristics of the component
- information on the grids
- information on the coupling/IO fields, also called 'transient variables'

5.3.1 Component model general characteristics

This type of information gives an overview of the component model:

- the component name: attribute 'local_name' of element 'prismcomponent', which should match the 2^{nd} argument of `PSMILe` call `prism_init_comp`(see section 4.1.2);
- a short general description of the component model: attribute 'long_name';
- the version of the OASIS4 Schema file: attribute 'oasis4_version';

[2] On the longer term, in order to avoid duplication of information, it is foreseen to develop a tool to extract automatically all PMIOD information which is already in the code (e.g. the component name given argument `comp_name` of `prism_init_comp` calls)

- the simulated part of the climate system: attribute 'simulated', either `ocean`, `sea_ice`, `ocean_biogeochemistry`, `atmosphere`, `atmospheric_chemistry`, or `land`;
- the name of the laboratory developing the component: element 'laboratory' in element 'code';
- the contact for additional information: element 'contact' in element 'code';
- the reference in the literature: element 'documentation' in element 'code';

5.3.2 Grid families and grids

This part contains information on the grids used by the component model. There might one or more grid families per component; for each grid family (element 'grid_family'), there may be one or more grids (elements 'grid'), each grid corresponding in fact to one resolution. All grids of all families shoule be described by the component developer in the PMIOD.

Each grid (element 'grid') is described by:

- the grid name: attribute 'local_name', which should match the 2^{nd} argument `grid_name` of `PSMILe` call `prism_def_grid` (see section 4.3.1)
- for the physical domain covered by the grid: element 'physical_space':
  - a general description: attribute 'long_name'
  - for the longitude dimension (element 'longitude_dimension'): the domain minimum and maximum and the units (elements 'valid_min', 'valid_max', and attribute 'units': for now, only `degrees_east` supported, see also section 4.3.2)
  - for the latitude dimension (element 'latitude_dimension'): the domain minimum and maximum and the units (elements 'valid_min', 'valid_max', and attribute 'units': for now, only `degrees_north` supported, see also section 4.3.2)
  - for the vertical dimension (element 'vertical_dimension'):
    - the domain minimum and maximum (element 'valid_min' and 'valid_max')
    - the units (attribute 'units': either `meters`, `bar`, `millibar`, `decibar`, `atmosphere`, `pascal`, `hPa`, `dimensionless`, see also section 4.3.2)
    - the direction in which the coordinate values are increasing (attribute 'positive', either `up` or `down`)
- for the sampled domain covered by the grid (element 'sampled_space'):
  - whether or not the grid covers the pole: attribute 'pole_covered', either `true` or `false`
  - the grid mesh structure type: attribute 'grid_type', which should match argument `grid_type` of `prism_def_grid` (see section 4.3.1), either:
    - `PRISM_gridless`
    - `PRISM_reglonlatvrt`
    - `PRISM_gaussreduced_regvrt`
    - `PRISM_irrlonlat_regvrt`,
    - `PRISM_unstructlonlat_regvrt`
    - `PRISM_unstuctlonlatvrt`
  - for each global index dimension (elements 'indexing_dimension'):
    - the index name: attribute 'local_name'
    - whether or not the grid is periodic is this dimension: attribute 'periodic' either `true` or `false`
    - the extent in this dimension: element 'extent'
    - the number of overlapping grid points in this index dimension: element 'nbr_overlap' (=0 if none)

- the computational space covered by the grid: element 'compute_space'. In the `PSMILe` , a grid is defined by its volume elements which discretize the domain covered. In these volume elements, a number of sets of points, on which the variables are calculated, can be placed. For vectors, three sets of points can be placed so that the vector components need not to be at the same location. Element 'compute_space' gives the user a description of the (vector) sets of points, declared in the component code:
  - elements 'points': the sets of points defined on the grid, declared in the code with `PSMILe` call `prism_set_points` (see section 4.3.2); for each set:
    - a local name which should match 2^{nd} argument in prism_set_point: attribute 'local_name'
    - a description of the set of points: attribute 'long_name'
  - elements 'vector': the vector sets, declared in the code with `PSMILe` call `prism_set_vector` (see section 4.3.8), which associates, for a vector variable, the 3 pre-defined sets of points on which the vector components are located; for each vector set:
    - a local name which should match 2^{nd} argument in `PSMILe` call `prism_set_vector` (attribute 'local_name')
    - a description of the vector (attribute 'long_ name')
    - the local name of the set of points on which the first, second, and third components are located (attribute 'firstcomp_points_local_name', 'secondcomp_points_local_name', and 'thirdcomp_points_local_name' respectively)

5.3.3 Coupling/IO fields (transient variables)

Each coupling/IO field possibly received or provided by the component model from/to its external environment (another model or a disk file) through `prism_get` or `prism_put` call should be described in the component PMIOD by one element 'transient' which has the following attributes and sub-elements:

- attribute 'local_name': the field name (which should match 2^{nd} argument in the corresponding `PSMILe` call `prism_def_var` , see section 4.4.1);
- attribute 'long_name': gives a general description of the variable;
- element 'transient_standard_name': the standard variable names following the CF convention (if it exist). This uniquely identifies the nature of the coupling/IO field. In case of vector, three elements need to be specified (one for each vector component).
- element 'physics': a description of the coupling/IO field physical constraints:
  - attribute 'transient_type': the coupling/IO field physical type (either 'single' or 'vector')
  - element 'physical_units': the coupling/IO field units
  - element 'valid_min': its physically acceptable minimum value
  - element 'valid_max': its physically acceptable maximum value
- element 'numeric', which attribute 'datatype' gives the coupling/IO field numeric type: either `xs:real`, `xs:double`, or `xs:integer`
- element 'computation', which attributes and sub-elements give some information on the coupling/IO field computational characteristics:
  - attribute 'maks', which tells whether or not a mask is associated to the coupling/IO field (either `true` or `false`)
  - attribute 'conditional_computation', which, if present, indicates under which condition the coupling/IO field is effectively sent and/or received
  - attribute 'method_type', which, if present, indicates what the coupling/IO field value represents on the grid cell, either `mean`, `max`, `min`, `median`, `variance`

  - element ‘associated_gridfamily’, which attribute ‘local_name’ must be the same than the one of the grid family associated to the coupling/IO field
  - element ‘associated_compute_space’, which attribute ‘local_name’ must be the same than the one of the computational space associated to the coupling/IO field (i.e. the attribute ‘local_name’ of either the associated set of points -element ‘points’-, or the associated vector sets - element ‘vector’)

- element ‘intent’, which describes if the coupling/IO field may be exported or imported, or both. The sub-elements of ‘intent’ are:
  - element ‘output’: if the coupling field can be exported through `PSMILe prism_put` call (see section 4.6.1), this element should contain the attribute ‘transi_out_name’ (a symbolic output name) and the element ‘minimal_period’, which is the period at which the `prism_put` is called in the code (to define this period the developer may specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements ‘nbr_secs’, ‘nbr_mins’, ‘nbr_hours’, ‘nbr_days’, ‘nbr_months’, ‘nbr_years’).
  - element ‘input’: if the coupling/IO field can be imported through a `prism_get` call (see section 4.6.2), this element should contain the element ‘minimal_period’, which is the period at which the `prism_get` is called in the code.

- element ‘transient_dependency’: if the developer wants to indicate a dependency between the coupling/IO field and another coupling/IO field from the same component, he has to define an element ‘transient_dependency’ and to specify this dependency in the attribute ‘dep_variable’. For example, field A is a transient_dependency of field B if field A is used in the calculation of field B. This information may be needed to prevent deadlocks.

5.4 The Specific Coupling Configuration (SCC)

The Specific Coupling Configuration (SCC) contains the general characteristics and process management information of one coupled model simulation. There must be one SCC per coupled model (or per standalone application), named `scc.xml`, and written by the coupled model user.

The SCC Schema is given in `/prism/src/mod/oasis4/util/xmlfiles/scc.xsd`.

After the call to `prism_init` in the application code, some of the SCC information is accessible directly by the model, with specific `PSMILe` calls (see section 4.2). In many cases, coherence with the compiling and running environment and scripts has to be ensured.

The SCC contains:

- the version of the OASIS4 Schema file: attribute ‘oasis4_version’ of element ‘scc’
- some general information on the experiment defined by the user (element ‘experiment’):
  - the experiment name (attribute ‘local_name’);
  - a description of the experiment (attribute ‘long_name’);
  - the mode into which all applications of the coupled model will be started (attribute ‘start_mode’: either `spawn` or `not_spawn`, see section 3.1); this user’s choice, restricted by the possibilities given in the ADs, determines the way the applications should be started in the run script.
  - the number of processes used for the OASIS4 Driver/Transformer (element ‘nbr_procs’ of element ‘driver’)
  - the start date of the experiment (element ‘start_date’)
  - the end date of the experiment (element ‘end_date’)
- some general information on the current run, which therefore must be updated for each run of the experiment (element ‘run’):

  - the start date of the run (element 'start_date'); the start date should correspond to the lower bound of the time interval which is represented by the first time step of the run.
  - the end date of the run (element 'end_date'); the end date should correspond to the upper bound of the time interval which is represented by the last time step of the run. Note that the end date of the current run has to be used as start date for the subsequent run.

- the list of applications chosen by the user (elements 'application'). For each chosen application:
  - the application name (as given in the corresponding AD) (attribute 'local_name') which must match argument `appl_name` of `PSMILe` call `prism_init` ;
  - the application executable name, defined by the compiling environment (attribute 'executable_name') (used only in `spawn` mode as argument of the `MPI_Comm_Spawn_Multiple`).
  - whether or not application stdout is redirect or not (user's choice) (attribute 'redirect', either `true` or `false`)
  - a list of launching arguments (chosen by the user in the list given in the corresponding AD) (elements 'argument');
  - a list of hosts (elements 'host'); for each host:
    * the host name (attribute 'local_name') (used only in `spawn` mode as argument of the `MPI_Comm_Spawn_Multiple`).
    * the number of processes to run this host (element 'nbr_procs') (used in the `not_spawn` method to split the global communicator; for the `spawn` method, used as argument in `MPI_Comm_Spawn_Multiple`).
  - the list of components activated (elements 'component', chosen by the user in the list given in the corresponding AD); for each component:
    * the component name (as given in the corresponding AD) (attribute 'local_name'), which must match the argument `comp_name` of `PSMILe` call `prism_init_comp` (see 4.1.2);
    * the lists of ranks in the total number of processes for the application (elements 'ranks'): The ranks are the numbers of the application processes (starting with zero) used to run the component model. They are given as lists of 3 numbers giving, in each list, a minimum value, a maximum value, and an increment value. For example, if processes numbered 0 to 31 are used to run a component model, this can be describe with one rank list (0, 31, 1); if processes 0 to 2 and 5 to 7 are used, this can be described with two rank lists (0, 2, 1) and (5, 7, 1).

5.5 The Specific Model Input and Output Configuration (SMIOC)

The Specific Model Input and Output Configuration (SMIOC) specifies the relations the component model will establish at run time with the rest of the coupled model through inputs and outputs for a specific run. It must be generated by the user for each component model based on the corresponding PMIOD information.

The SMIOC Schema is given in `/prism/src/mod/oasis4/util/xmlfiles/smioc.xsd`. The SMIOC file name must be <application_local_name>_<component_local_name>_smioc.xml where <application_local_name> is the application 'local_name attribute and <component_local_name> is the component 'local_name' attribute in the scc.xml file. Examples of SMIOC xml files for the toy coupled model TOYOA4 can be found in `prism/util/running/toyoa4/input`.

The SMIOC may contains 3 types of information detailed in the next paragraphs:

- general characteristics of the component, as described in the corresponding PMIOD
- information on the grids
- information on the coupling/IO fields, also called 'transient variables'

As stated below, the description information of the corresponding PMIOD may be repeated in the SMIOC. Part of this description information is used to define attributes of the I/O NetCDF files but is not mandatory for the proper execution of the coupled model *per se*; if it is not specified in the SMIOC, it will just be missing in the I/O files. In the paragraphs below, it is detailed which information is mandatory.

5.5.1 Component model general characteristics

The SMIOC may repeat the description information provided about the component model general characteristics in the corresponding PMIOD. For more detail, see section 5.3.1. However, the only mandatory information about the component model general characteristics in the SMIOC is the component name, i.e. the attribute 'local_name' of element 'prismcomponent', which must match the 2^{nd} argument of `PSMILe` call `prism_init_comp`(see section 4.1.2), and the attribute 'oasis4_version' of element 'prismcomponent'.

5.5.2 Grid families and grids

This part contains information on the grids effectively used during the run by the component model, based on the description done in the corresponding PMIOD file.

There might one or more grid families per component as described in the corresponding PMIOD. But for each grid family (element 'grid_family'), only one grid, i.e. in fact one resolution, can now be specified in the SMIOC.

For each grid family, the chosen grid (element 'grid') can be described in the SMIOC as in the PMIOD (see section 5.3.2). However, the only mandatory grid information in the SMIOC is:

- the grid name: attribute 'local_name', which must match the 2^{nd} argument `grid_name` of `PSMILe` call `prism_def_grid` (see section 4.3.1).
- for the sampled domain covered by the grid (element 'sampled_space'):
  - whether or not the grid covers the pole: attribute 'pole_covered', either `true` or `false`
  - the grid mesh structure type: attribute 'grid_type', which must match argument `grid_type` of `prism_def_grid` (see section 4.3.1), either:
    - `PRISM_gridless`
    - `PRISM_reglonlatvrt`
    - `PRISM_gaussreduced_regvrt`
    - `PRISM_irrlonlat_regvrt`,
    - `PRISM_unstructlonlat_regvrt`
    - `PRISM_unstuctlonlatvrt`
  - for each global index dimension (elements 'indexing_dimension'):
    - the index name: attribute 'local_name'
    - whether or not the grid is periodic is this dimension: attribute 'periodic' either `true` or `false` (mandatory only if the grid is periodic)

5.5.3 Coupling/IO fields (transient variables)

Each coupling/IO field effectively received or provided by the component model from/to its external environment (another model or a disk file) through `prism_get` or `prism_put` call in the component code (see sections 4.6.1 and 4.6.2) must be specified by one element 'transient' which has the following attributes and sub-elements:

- attribute 'local_name': the field name, which must match 2^{nd} argument in the corresponding `PSMILe` call `prism_def_var` (see sections 4.4.1); (mandatory);

- attribute ‘long_name’: gives a general description of the variable; (optional)
- element ‘transient_standard_name’: one or more PRISM standard names following the CF convention (if they exist); see section 5.3.3 for details; (mandatory)
- element ‘physics’: a description of the coupling/IO field physical constraints; see section 5.3.3 for details; (optional)
- element ‘numeric’, which attribute ‘datatype’ gives the coupling/IO field numeric type (either `xs:real`, `xs:double`, or `xs:integer`); (mandatory)
- element ‘computation’, which attributes and sub-elements give some information on the coupling/IO field computational characteristics; see section 5.3.3 for details; (optional)
- element ‘intent’, which describes if the coupling/IO field will be exported or imported, or both (mandatory). This element contains in its sub-elements all coupling and I/O information (source and/or target, frequency, transformations, interpolation, etc.). The sub-elements of ‘intent’ are:
  - element ‘output’: If the coupling/IO field is exported through a `prism_put` , it can be effectively be sent to none, one, or many targets; each target must be described in one element ‘ouput’. The element ‘output’ is described in more details in section 5.5.4.
  - element ‘input’: If the coupling/IO field is imported through a `prism_get` , this import must be described in one element ‘input’. The element ‘input’ is described in more details in section 5.5.5.
- element ‘transient_dependency’: optional. See section 5.3.3 for details.

5.5.4 The ‘output’ element

If the coupling/IO field is exported through a `prism_put` in the component code, it can be effectively be sent to none, one, or many targets, each target being described in one element ‘ouput’. A more detailed description of element ‘output’, its attributes and sub-elements is given here.

1. attribute ‘transi_out_name’: a symbolic name defined by the user for that specific ‘output’ element.
2. element ‘minimal_period’: The period at which the `prism_put` is called in the code; this element should be specified as in the corresponding PMIOD file, if it exists. To define this period the developer may specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements ‘nbr_secs’, ‘nbr_mins’, ‘nbr_hours’, ‘nbr_days’, ‘nbr_months’, ‘nbr_years’.
3. element ‘exchange_date’: The dates at which the coupling or I/O will effectively be performed. To express these dates, the user has to specify one of the following sub-elements:
   - element ‘period’: The coupling or I/O is performed with a fixed period. To define this period, the user may specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements ‘second’, ‘minute’, ‘hours’, ‘day’, ‘month’, ‘year’.
4. element ‘corresp_transi_in_name’: The symbolic name of the corresponding input coupling/IO field origin (attribute ‘transi_in_name’ of element ‘origin’ of element ‘input’) in the target component or target file. This defines an exchange between a source and a target component or file. Coherence has to be ensured, i.e. the value of the current output ‘transi_out_name’ attribute (see above) has to be specified in the ‘corresp_transi_out_name’ element of the corresponding input coupling field origin (see also section 5.5.5). When it will be available, this coherence will be automatically ensured by the GUI.
5. element ‘file’ or element ‘component_name’: The target file description (I/O) or the target component ‘local_name’ attribute (coupling). The ‘file’ element is described in more detail in section 5.5.7.

6. element 'lag': The number of `prism_put` periods[3] to add to the output coupling field `prism_put` date and date_bounds to match the corresponding input coupling field `prism_get` date in the target component (see also 4.6.4).
7. element 'source_transformation': The transformations performed on the output coupling/IO field in the source component `PSMILe` .
   - element 'source_time_operation': for each grid point, the output coupling/IO field can be averaged (`taverage`) or accumulated (`accumul`) over the last coupling period by the source `PSMILe` and the result is transfered.
   - element 'statistics': different statistics (minimum, maximum, integral) are calculated for the field on the masked points, and/or on the not masked points, and/or on all points of the output coupling/IO field, if respectively the sub-elements 'masked_points', and/or 'notmasked_points', and/or 'all_points' are specified. This is done below the `prism_put` by the source `PSMILe` (after the time operations described in element 'source_time_operation' if any). These statitistics are printed to the `PSMILe` log file for information only; they do not transform the output coupling/IO field.
   - element 'source_local_transformation': the following local transformations may be performed on the output coupling/IO field by the source `PSMILe` :
     - element 'scattering': the 'scattering' should be specified by the developer in the PMIOD and should not be changed by the user in the SMIOC. It is performed on an output coupling/IO field below the `prism_put` by the source `PSMILe` . It is required when grid information transfered to the `PSMILe` includes the masked points and when the array transfered to the `prism_put` API is a vector gathering only the non-masked points.
     - element 'add_scalar': The scalar specified in this element is added to each grid point coupling/IO field value.
     - element 'mult_scalar': Each grid point coupling/IO field value is multiplied by the scalar specified in this element.
8. element 'debug_mode': either `true` or `false`; if it is `true`, the output coupling/IO field is automatically also written to a file below the `prism_put` .

5.5.5 The 'input' element

If the coupling/IO field is imported through a `prism_get` in the component code, the user describes one source for that field in the SMIOC. A more detailed description of element 'input', its attributes and sub-elements is given here.

1. attribute 'required_but_changeable': if the developer indicates in the PMIOD that this attribute is `true`, the user must define at least one 'input' element in the SMIOC; if it is `false`, then an 'input' with no 'origin' sub-elements may appear in the SMIOC.
2. element 'minimal_period': The period at which the `prism_get` is called in the code. (See element 'minimal_period' of element 'output' in section 5.5.4.)
3. element 'exchange_date': The dates at which the coupling or I/O will effectively be performed (see 'exchange_date' in 'output' in section 5.5.4).
4. element 'origin': In the current OASIS4 version, an input coupling/IO field may come only from one origin being described by an element 'origin' which contains the following attributes and sub-elements:
   - attribute 'transi_in_name': a symbolic name defined for that specific 'origin' element.

[3] A `prism_put` period is the time between the `prism_put` date_bounds; e.g. for a lag of 1, the time added to the `prism_put` date and date_bounds arguments would be once the time difference between the associated date_bounds.

- element 'corresp_transi_out_name': The symbolic name of the corresponding output coupling/IO field (attribute 'transi_out_name' of element 'output') in the source component or source file. This defines an exchange between a source and a target component or file. Coherence has to be ensured, i.e. the value of the current input 'transi_in_name' attribute has to be specified in the 'corres_transi_in_name' element of the corresponding output coupling field (see also section 5.5.4). When it will be available, this coherence will be automatically ensured by the GUI.
- element 'file' or 'component_name': The source file description (I/O) or the source component 'local_name' attribute (coupling). The 'file' element is described in more detail in section 5.5.7.
- element 'middle_transformation': The transformations which link the source and the target.
  - element 'interpolation': The interpolation to be performed on the output coupling field to express it on the target model grid. This element is described in more detail in section 5.5.6.

5. element 'target_transformation': The transformations performed on the input coupling/IO field in the target component `PSMILe` .
   - element 'target_local_transformation': The local transformations performed on the input coupling/IO field.
     - element 'gathering': The 'gathering' should specified by the developer in the PMIOD and should be kept as is in the SMIOC. It is performed on an input coupling/IO field below the `prism_get` by the target `PSMILe` . It is required when the grid information transfered to the `PSMILe` covers the whole grid (masked points included), and when the array transfered through `prism_get` API is a vector gathering only the non-masked points.
     - element 'add_scalar': The scalar specified in this element is added to each grid point coupling/IO field value.
     - element 'mult_scalar': Each grid point coupling/IO field value is multiplied by the scalar specified in this element.
   - element 'target_time_operation': Target time interpolation is supported below the `prism_get` only for IO data[4]. The types of time interpolation are the nearest neighbour 'time_nneighbour' and linear time interpolation between the two closest timestamps 'time_linear' in the input file.
   - element 'statistics': see section 5.5.4.
6. element 'debug_mode': either `true` or `false`; if it is `true`, the input coupling/IO field is automatically written to a file below the `prism_get` .

5.5.6 The element 'interpolation'

The element 'interpolation' is a sub-element of 'middle_transformation', which is a sub-element of 'origin', which is a sub-element of 'input'. The interpolation is needed to express the coupling field on the target model grid[5].

As all coupling arrays are given on a 3D grid, the user has to choose among the following:

[4]This feature is not essential for coupling data as each `prism_put` has a date and date_bounds as arguments. Therefore, a `prism_put` and a `prism_get` will be matched if the `prism_get` date falls into the date_bounds of the `prism_put` . Allowing for time interpolation, e.g. allowing a `prism_get` to match with an averaged value of the two `prism_put` nearest neighbour in time, could lead to deadlocks as the model performing the `prism_get` would be blocked until the two `prism_put` nearest neighbour in time are performed. We rely only the date_bounds to match `prism_put` and `prism_get` having non matching dates.

[5]In the current OASIS4 version, interpolation is available only for coupling fields. In a future version, interpolation might also be possible for I/O fields read/written from/to a file.

- 'interp3D': A full 3D interpolation.
- '(interp2D, interp1D)': The same 2D interpolation for all vertical levels followed by a 1D interpolation in the vertical[6] This type of interpolation can be used for all grids which vertical dimension can be expressed as z(k), i.e. for i.e. for source grid types `PRISM_reglonlatvrt`, or `PRISM_irrlonlat_regvrt`. The mask may vary with depth. Currently the combination of 2D and 1D interpolations that are supported are `bilinear` and `none`, `nneighbour2D` and `none` (see below).

The elements 'interp3D', 'interp2D', 'interp1D', are separately described here after:

1. element 'interp3D': For 3D interpolation, the user has to choose among the following methods:
   - element 'nneighbour3D': A 3D nearest neighbour algorithm; the parameters are:
     - element 'para_search': currenlty, only `local` is available (a local but less expensive neighborhood search).
     - element 'nbr_neighbours': the number of neighbours.
     - element 'used_masked': either `true` (all points are considered in the `PSMILe` neighbourhood search and the Transformer detects masked points), or `false` (the nearest neighbours are chosen by the `PSMILe` among non-masked points only).
   - element 'trilinear': A trilinear algorithm; the parameters are:
     - element 'para_search': see element 'nneighbour3D' above.
     - element 'if_masked': either `novalue`, `tneighbour`, or `nneighbour`.
       * `novalue`: if some of the 8 trilinear neighbours are masked, `PRISM_undef` value is given to that target point;
       * `tneighbour`: if some of the 8 trilinear neighbours are masked, the non-masked points among those 8 points are used for calculating a weighted average; if the 8 trilinear neighbours are masked, `PRISM_undef` value is given to that target point;
       * `nneighbour`: if some of the 8 trilinear neighbours are masked, the non-masked points among those 8 points are used for calculating a weighted average; if the 8 trilinear neighbours are masked, the non-masked nearest neighbour is used.
2. element 'interp2D': For 2D interpolation, the following methods can be chosen:
   - element 'nneighbour2D': A 2D nearest neighbour algorithm; the parameters are:
     - elements 'para_search', 'nbr_neighbours', 'used_masked': see element 'nneighbour3D' above.
   - element 'bilinear': A bilinear algorithm; for the parameters are:
     - element 'para_search': see element 'nneighbour3D' above.
     - element 'if_masked': see element 'trilinear' above.
   - element 'bicubic': A bicubic algorithm, the parameters are:
     - element 'para_search', 'if_masked', 'gradient_varname': see above.
     - element 'bicubic_method': The bicubic method: either `gradient` (the four enclosing source neighbour values and gradient values are used), or `sixteen` (the sixteen enclosing source neighbour values are used).
3. element 'interp1D' For 1D interpolations, the following methods can be chosen:
   - element 'none':

     Interpolation method that can be chosen for dimension with extent of 1. For example, to interpolate a field of Sea Surface Temperature dimensioned (i,j,k) with extent of k being 1,

[6] Currently, only the `none` interpolation, i.e. no interpolation, is available in the vertical.

the interpolation type can be '(interp2D, interp1D)' and 'none' should be chosen for the 'interp1D'.

5.5.7 The 'file' element

The 'file element is composed of the following sub-elements:

- element 'name': a character string used to build the file name.
- element 'suffix': either `true` or `false`. If 'suffix' is false (by default), the file name is composed only of element 'name'; if it is true, the file name is composed of element 'name' to which the PRISM suffix for dates is added. When the file is opened for writing, the suffix will be "_out.<job_startdate>.nc", where <job_startdate> is the start date of the job. When the file is opened for reading, the suffix should be "_in.<start_date>.nc", where <start_date> is the date of the first time stamp in that file. When reading an input from a file, the `PSMILe` will automatically match the requested date of the input with the appropiate file if it falls into the time interval covered by that file. The <job_startdate> and <start_date> must be written according to the ISO format yyyy-mm-ddTHH:MM:SS. The date/time string in the file name must have to format yyyy-mm-ddTHH.MM.SS since the colon is already used in other context for file systems.
- element 'format': the format of the file; only NetCDF (`mpp_netcdf`) supported for now.
- element 'io_mode': either `iosingle` (by default) or `distributed`. The mode `iosingle` means that the whole file is written or read only by the master process; `distributed` means that each process writes or reads its part of the field to a different partial file. Note that if the PSMILe is linked against the parallel NetCDF library (7), the `parallel` mode will automatically be used; in this case each process writes its part of the field to one parallel file (see also our remarks about parallel NetCDF on page 24).
- element 'packing': packing mode , either `1`, `2`, `4` or `8` (for NetCDF format only)
- element 'scaling': if present, the field read from the file are multiplied in the `PSMILe` by the 'scaling' value (1.0 by default) (for NetCDF format only)
- element 'adding': if present, the 'adding' value (0.0 by default) is added to the field read from the file (for NetCDF format only)
- element 'fill_value': on output, specifies the value given to grid points for which no meaningfull value was calculated; on input, specifies the value given in the file to undefined or missing data.

Chapter 6

Compiling and running OASIS4 and TOYOA4

This chapter describe how to compile and run the OASIS4 coupler and its toy coupled model “TOYOA4”.

6.1 Introduction

The current OASIS4 version was successfully compiled and run on the following platforms:

- Intel(R) Xeon(TM) Infiniband Cluster
- Intel(R) Xeon(TM) Myrinet Cluster
- Linux PC DELL Precision 380 (Pentium 4, 3.2 Ghz)
- NEC SX6 and SX8
- SGI O3000/2000 server with MIPS 4 processors and IRIX64
- SGI IA64 Linux server Altix 3000 and Altix 4000 (under Redhat AS3/AS4 and SuSE, SLES9, SLES 10)
- IBM Cluster 1600 (IBM Power4)

with the following Fortran Compilers:

- Intel Fortran Compiler Version 9.0 64Bit and 32Bit
- Portland Group Compiler Version 6.1-6 32-bit
- NEC SX Fortran Compiler.
- SGI MIPSPro compiling system 7.3 and 7.4
- Intel compiling systems ifort and icc 9.1, 9.0 and 8.1
- IBM XL Fortram Compiler

6.2 Compiling OASIS4 and its associated PSMIle library

Compiling can be done using either a top makefile `TopMakefileOasis4` and platform dependent header files (see section 6.2.1) or the PRISM Standard Compile Environment (SCE) (see section 6.2.2). For both methods, the same low-level makefiles in each source directory are used. During compilation, a new directory branch is created `/prism/`*arch*, where *arch* is the name of the compiling platform architecture (e.g. *Linux*). After successful compilation, resulting executables are found in `/prism/`*arch*`/bin`, libraries in `/prism/`*arch*`/lib` and object and module files in `/prism/`*arch*`/build`.

6.2.1 Compilation with TopMakefileOasis4

Compiling OASIS4 and TOYOA4 using the top makefile `TopMakefileOasis4` can be done in directory `prism/src/mod/oasis4/util/make_dir`. `TopMakefileOasis4` must be completed with a header file `make.`*your_platform* specific to the compiling platform used and specified in `prism/src/mod/oasis4/util/make_dir/make.inc`. One of the files `make.pgi_cerfacs`, `make.sx_frontend` or `make.aix` can by used as a template. The root of the prism tree can be anywere and must be set in the variable `PRISMHOME` in the `make.`*your_platform* file. The choice of MPI1 or MPI2 is also done in the `make.`*your_platform* file (see `CHAN` therein).

The following commands are available:

- `make -f TopMakefileOasis4`
  compiles OASIS4 libraries *common_oa4*, *psmile_oa4* and *mpp_io* and creates OASIS4 Driver/Transformer executable oasis4.MPI[1/2].x ;
- `make toyoa4 -f TopMakefileOasis4`
  compiles OASIS4 libraries as above and creates OASIS4 and TOYOA4 executables oasis4.MPI[1/2].x, atmoa4.MPI[1/2].x, oceoa4.MPI[1/2].x and lanoa4.MPI[1/2].x ;
- `make help -f TopMakefileOasis4`
  displays help information ;
- `make clean -f TopMakefileOasis4`:
  cleans OASIS4 and TOYOA4 compiled files, but not the libraries ;
- `make realclean -f TopMakefileOasis4`:
  cleans OASIS4 and TOYOA4 compiled files including libraries.

Log and error messages from compilation are saved in the files COMP.log and COMP.err in make_dir.

For not compiling the mpp_io library, the variable `LIBMPP` must be left undefined in the file make.*your_platform*; in this case, the top makefile activates the CPP key `key_noIO` and only empty mpp_io files are compiled.

6.2.2 Compilation using the PRISM Standard Compiling Environment (SCE)

The PRISM Standard Compiling Environment (SCE) has been adapted for OASIS4. These modifications are available on CERFACS CVS server `alter` and will also be included in the next official release of the SCE on the new PRISM Subversion server at DKRZ in Hamburg.

Scripts and include files for the SCE are found in directory branch `prism/util/compile` (see figure 6.1). The toy model TOYOA4 using OASIS4 has been successfully compiled and run for the 3 platforms currently included in the SCE available from CERFACS CVS server: the NEC SX6 at DKRZ (*nodename* = `ds`, see `/frames/include_ds`), the IBM power4 at ECMWF (*nodename* = `hpc`, see `/frames/include_hpc`) and the Linux PC at CERFACS (*nodename* = `kullen`, see `/frames/include_kullen`).

For compiling OASIS4 and TOYOA4 within SCE, the compilation scripts first have to be created by using `Create_COMP_cpl_models.ksh` in `prism/util/compile/frames`:

`./Create_COMP_cpl_models.ksh toyoa4 [`*expid*`] [`*nodename*`] [MPI1` *or* `MPI2]`

where the last 3 arguments are optional.

This will create 4 model compilation scripts:

- `prism/src/mod/atmoa4/COMP_atmoa4_`*expid.nodename*
- `prism/src/mod/lanoa4/COMP_lanoa4_`*expid.nodename*
- `prism/src/mod/oasis4/COMP_oasis4_`*expid.nodename*
- `prism/src/mod/oceoa4/COMP_oceoa4_`*expid.nodename*

and 1 library compilation script:
- `prism/util/COMP_libs.`*node_name*.

Then each model compilation script has to be executed in its directory; the library compilation script is executed automatically by each of the model compilation script.

During compilation, log and error messages are written into files with suffix .log and .err in the same directory than the compilation script. Log and error messages after compilation of the libraries are found in `prism/util/COMP_libs.log` and `COMP_libs.err`.

For compiling without mpp_io library, the variable `use_key_noIO` has to be changed to "yes" in the compile scripts for atmoa4, oceoa4 and lanoa4; in that case, only empty mpp_io files are compiled.

6.2.3 Some details on the compilation

- Other librairies needed

  The following librairies (not provided with the OASIS4 sources) are required:
  - Message Passing Interface, MPI1 (9) or MPI2 (5) (MPICH, SGI native MPI, NEC SX native MPI, LAM-MPI and SCAMPI were successfully tested)
  - NetCDF Version 3.4 or higher (4) or parallel NetCDF Version 1.0.0 (7) (see page 4.6)
  - libxml Version 2.6.5 or higher (10)
- CPP keys

  The following CPP keys can be activated:
  (see `CPPDEF` in `prism/src/mod/oasis4/util/make_dir/make.xxx` files or `OSspecific_`*nodename*`.h` in `prism/util/compile/frames/include_`*nodename*)
  - PSMILE_WITH_IO: to make use of the IO capability of `PSMILe`
  - PRISM_WITH_MPI1: This options has to be chosen if the available MPI library supports MPI1 standard, like mpich1.2.* or does not support the full MPI2 standard.
  - PRISM_WITH_MPI2: When the available MPI2 library supports the complete MPI2 standard this option may be chosen instead.
  - PRISM_LAM: if LAM-MPI library is used.
  - DONT_HAVE_ERRORCODES_IGNORE: As a workaround for some MPI2 implementations that do not support the MPI parameter MPI_ERRORCODES_IGNORE this key has to be activated. If at all, it is only needed in conjunction with PRISM_WITH_MPI2.
  - SX: To achieve better performance on vector architecture this option should be set.
  - VERBOSE: Useful for debugging purposes activation this key will cause the library and driver routines to run in verbose mode. Since all output is immediately flushed to standard output this will significantly decrease performance and is therefore not recommended for production runs.
  - DEBUG: Activating this option will cause the driver and library to write out additional output for debugging purpose. This output is immediately flushed to standard output and will therefore decrease performance.
  - PRISM_ASSERTION: Mainly used by the developers; the code encapsulated by this cpp key will perform additional internal consistency checks and will provide additional information for debugging.

6.2.4 Remarks and known problems

- LAM-MPI with the `spawn` approach

The usage of `MPI_Comm_Spawn_Multiple` is the most portable way if MPI processes shall be dynamically spawned on multiple hosts. Therefore, there is a reserved predefined key "host" for the info argument, which specifies the value of the host name, in the MPI2 standard. Nevertheless this is currently not supported by LAM-MPI. Therefore, to use LAM-MPI, it is required to use the CPP key PRISM_LAM. In this case, LAM-MPI `MPI_Comm_Spawn_Multiple` fills the processors according to the list given in the `lam.config` file used by the lamboot process (see example in `PRISM_Cpl/examples/simple-mg`), using always all processors on a given node. For example, 1 Driver/Transformer process and 4 processes for the ocean and the atmosphere models would be scheduled on three 4-CPU hosts like the following: the Driver/Transformer would be on host 1, the ocean model would have 3 processes on host1 and 1 process on host 2, and the atmosphere model would have 3 processes run on host 2 and 1 on host 3, which of course is not optimal.

With `MPI_Comm_Spawn` , LAM-MPI would be more more flexible regarding the spawning of processes. For OASIS4 this is not an option since `MPI_Comm_Spawn_Multiple` is required for

- starting multiple binaries (not several applications); this may be required for an heterogenous cluster;
- starting same binary with a multiple set of arguments;
- placing multiple binaries in the same MPI_COMM_WORLD. It is intended by PRISM to place the MPI processes of an application into a MPI_COMM_WORLD which is different for each application. In this case, the applications are not required to change the application internal communicators.

Therefore, the `spawn` approach is not recommended with LAM-MPI. The `not_spawn` approach (see sections 3.1) should be prefered if possible.

- MPICH

  Since MPI1 is not designed for 64 Bit architectures the default MPICH.1.2.* implementation will not work on 64 Bit systems for OASIS4 and `PSMILe`. It could work on IA64 if there was no use of functions with INTEGER arguments representing an address or a displacement as is the case in OASIS4 (on IA64 architectures these integers must be 64 bits or "long" in C language; they are "int" in MPICH) .

- Portland Group Compiler

  The Portland Group Compiler Version 5.2 produces an internal compiler error for the main routine of OASIS4.

  For the Portland Group Compiler Version 6.0, the debug option (-g) must be used. No particular option is needed for version 6.1 .

  The Portland Group C compiler produces an error. Use of GNU C compiler gcc is recommended instead (see `CC` in `prism/src/mod/oasis4/util/make_dir/make.xxx` files or `Sitespecific_`*nodename*`.h` in `prism/util/compile/frames/include_`*nodename*)

- Intel Fortran Compiler

  To successfully compile OASIS4 Intel Fortran Compiler version 8.0 or higher is required.

6.3 Running TOYOA4

Input files, data and script for running TOYOA4 are found in prism/util/running/toyoa4, see figure 6.1. Note that TOYOA4 has not been adapted to PRISM Standard Running Environment.

NetCDF data files needed for running TOYOA4 are found in directory `/data`. The description and configuration XML files are found in directory `/input`. Running can be done with a run script `run_toyoa4` in directory `/script` which first will create the working directory `/work`; all files and executables needed for running are first copied into this working directory. The run script `run_toyoa4` was run

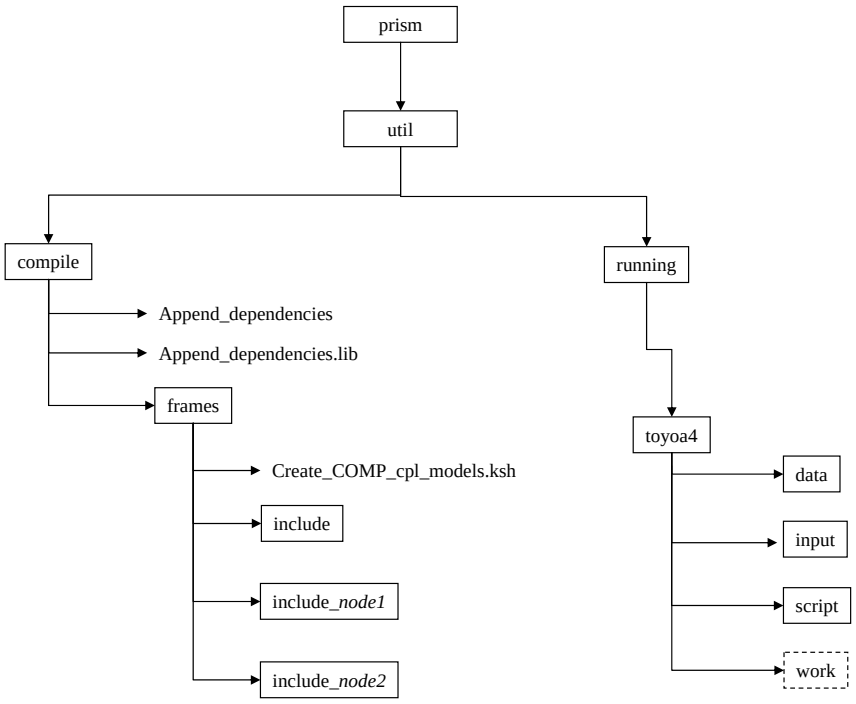


Figure 6.1: Directories in prism/util

on three platforms, Linux at CERFAXS, SX6 at DKRZ and IBM power4 at ECMWF, using MPI1 (which means that OASIS4 and 3 component model executables are started in the script). The script `run_toyoa4` is an example of running TOYOA4 and can be modified by the user for his/her platform.

Figure 6.2 illustrates the coupling and I/O exchanges occuring between the 3 toy component models atmoa4, oceoa4, and lanoa4.

Both atmoa4 and lanoa4 work on a T31 Gaussian grid, but their parallel partitioning is a function of their number of processes which can be different. The third model, oceoa4, is not parallel and uses a a real ocean model cartesian, stretched and rotated grid of 182X149 grid points.

All coupling and I/O fields are scalar fields. The model atmoa4 declares 1 input field `SISUTESU`, and 4 output field `CONSFTOT, COSENHFL, COWATFLU, ATWINSTS` as is listed in its PMIOD file `atmoa4_atmos_pmiod.xml`. The model lanoa4 declares 2 input fields `LAWATFLX` and `SOSENHFL`, and 1 output field `LARUNOFF` as is listed in its PMIOD file `lanoa4_land_pmiod.xml`. The model oceoa4 declares 4 input fields `SONSHLDO, SOWAFLDO, SORUNOFF` and `OCWINSTS`, and 1 output field `SOSSTSST`.

At run-time, the OASIS4 Driver/Transformer and the PSMILe model interface linked to the component models act according to the specifications written by the user in the configuration SMIOC XML files.

In the atmoa4 SMIOC file `atmoa4_atmos_smioc.xml`, it is specified that `ATWINSTS` will be sent to oceoa4, `COSENHFL` to lanoa4, `COWATFLU` both to oceoa4 and lanoa4, while `CONSFTOT` is not sent at all; it is also specified that `SISUTESU` will come from oceoa4. The lanoa4 SMIOC file `lanoa4_land_smioc.xml` specifies that `LARUNOFF` will both go to oceoa4 and be written to a file `LARUNOFF.nc` and that `LAWATFLX` and `SOSENHFL` will be received from atmoa4. Finally, in the

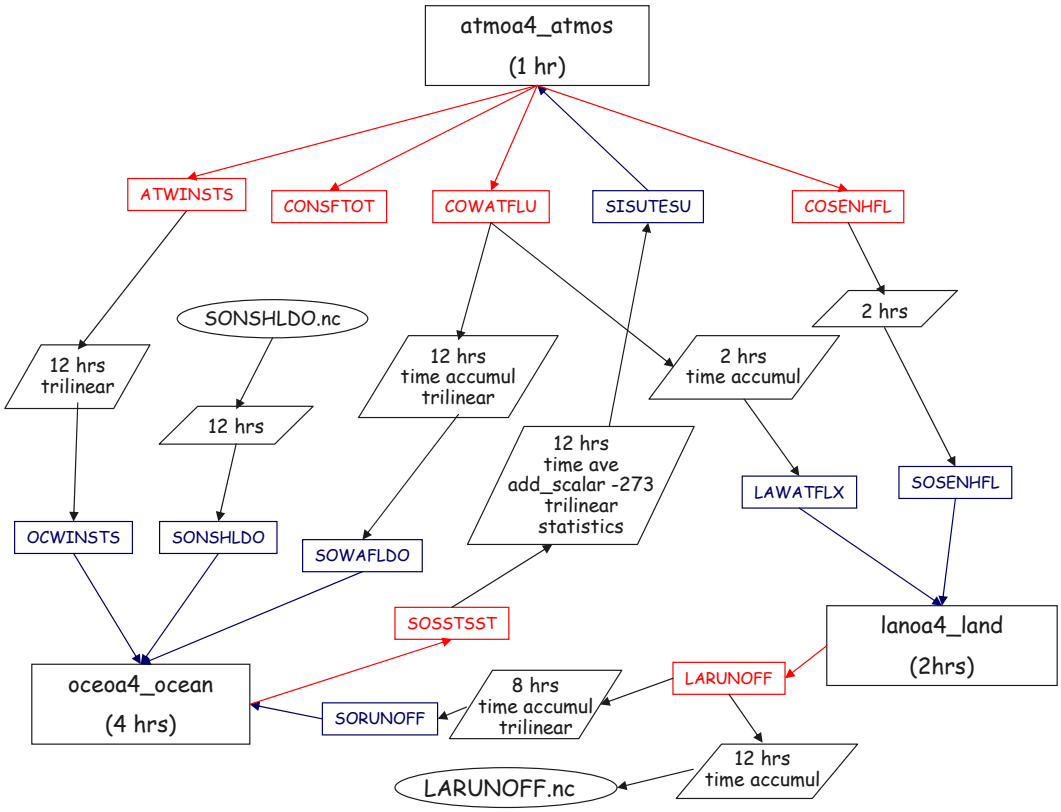


Figure 6.2: TOYOA4 toy coupled model coupling and I/O configuration

oceoa4 SMIOC file `oceoa4_ocean_smioc.xml`, it is specified that `OCWINSTS` and `SOWAFLDO` will be received from atmoa4, `SORUNOFF` from lanoa4, while `SONSHLDO` will be read from a file `SONSHLDO.nc`; `SOSSTSST` will be sent to atmoa4.

Different operations are performed by the PSMILe model interface on the coupling or I/O fields such as statistics, time accumulation time averaging, as specified in the SMIOC files. The exchanges of the coupling fields between atmoa4 and lanoa4 (and vice-versa) are direct, involving possibly some repartitioning if their parallel partitioning are different. As atmoa4 and oceoa4 do not have the same grid, their exchanges of coupling fields go through the Transformer (not illustrated on figure 6.2) where a linear interpolation is performed. The different coupling and I/O periods are also specified in the different SMIOC files.

TOYOA4 also illustrates the use of a coupling restart file for field `COSENHFL` for which a positive lag of 2 is defined. The first time TOYOA4 is run, the variable `run` should be set to `start` in `run_toyoa4`. In that case, the file `scc.xml.start` is copied in `scc.xml` and used, TOYOA4 is run for 3 days starting January 1^{st} 2000, and the first field `COSENHFL` received by lanoa4 comes from the restart file `COSENHFL_toyatm_atmos_rst.2000-01-01T00_00_00.nc`; at the end of the run, the restart file for the next run, `COSENHFL_toyatm_atmos_rst.2000-01-04T00_00_00.nc`, is created by the last call to prism_put for `COSENHFL` in atmoa4. A next run of 3 days starting January 4^{th} 2000 can then be run by changing `run=restart` in `run_toyoa4` and executing `run_toyoa4` again.

A successfull execution of TOYOA4 (with `run` set to `start` in `run_toyoa4`) produces files that can be compared to results in `prism/util/running/toyoa4/outdata`. In particular, files containing standard output from the different components (e.g. atmoa4.0, lanoa4.0, oceoa4.0) should end with lines like

```
 -------------------------------------
 --- Note: MPI_Finalize was called ---
 ---       from prism_terminate.   ---
 -------------------------------------
```

Appendix A

Scalability with OASIS4

One of the major enhancements of OASIS4 compared to OASIS3 is the full parallelization of the `PSMILe` (see section 4.5.1) and the Transformer (see section 3.2).

In 2004, at the end of the EU PRISM project funded by the European Community, the toy coupled model `simple-mg` (see directory `prism/src/mod/oasis4/src/examples`) with a T106 resolution for the atmosphere toy model was selected for scalability tests. The results of those scalability tests are presented here, even if they were not performed with the current OASIS4 version.

Selected platforms were NEC SX-6, SGI Altix and Origin, AMD-Athlon PC Cluster, and AMD-Opteron PC Cluster. Table A.1 summarizes the characteristics of the tested systems and used software.

Model/Feature	CPU specs	Main Memory (per CPU)	Compiler	MPI-library
NEC SX-6	0,5 GHz (*16)	8 GB	F90: Rev. 285 C++: Rev.063	NEC-MPI LC310039
SGI ALTIX	Madison 1,5 Ghz 6 MB L3-cache	2 GB	Intel ifort 8.050 Intel icc 8.069	SGI MPT 1.12
SGI-Origin	R14000 0,7 Ghz 8 MB L2-cache	2 GB	MIPSPro 7.4.1	SGI MPT 1.12
AMD-Athlon PC	2,8 GHz, 32 bit	4 GB	Absoft 32bit F95 9.0 r2	MPICH-Myrinet
AMD-Opteron PC	2,2 GHz, 64 bit	4 GB	Pathscale 1.4.1	LAM 7.1.1

Table A.1: Characteristics of the tested systems and used software for scalability tests

Simulation with up to 24 CPUs were carried out, starting with one process for each component model and the Transformer (1-1-1) and ending with 8 processes per component model and the Transformer (8-8-8). The notation in the result tables below is X-Y-Z where X, Y, and Z are respectively the number of processes for the atmosphere toy model, for the Transformer, and for the ocean toy model. For example: 4-1-4 means 4 processes for each component model and 1 processes for the Transformer.

Two measures of the scalability is taken in each `simple-mg` toy component model:

- the time in seconds until the `prism_enddef` is reached. This measure is reported in the columns ‘enddef ATM’ and ‘enddef OCE’ for respectively the atmosphere and the ocean component model in the tables below. The subroutine `prism_enddef` (see section 4.5.1) finishes the definition phase and includes the parallel neighborhood search for the interpolation done in parallel by the `PSMILe` linked to the models. Increasing the number of processes for the component models should therefore reduce this time. This is a measure of the `PSMILe` scalability.
- the required time in seconds for a ping-pong exchange of data with the other component. This measure is reported in the columns ‘ping-pong ATM’ and ‘ping-pong OCE’ for respectively the atmosphere and the ocean component model in the tables below. As the data are transfered in

parallel by the PSMILe and interpolated by the Transformer, increasing the number of processes for the Transformer and for the component models should reduce this time. This is a measure of the Transformer and of the PSMILe scalability.

SX6	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1 (1 node)	0.6	0.6	0.4	0.4
2-2-2 (1 node)	0.4	0.4	0.2	0.2
4-4-4 (2 nodes)	0.4	0.2	0.05	0.2

Table A.2: Scalability results for simple-mg on NEC SX-6.

SGI	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	3.5	1.8	1.1	1.9
2-2-2	1.6	1.3	0.6	1.0
4-4-4	0.7	0.6	0.3	0.5
4-1-4	1.0	0.9	0.8	0.8

Table A.3: Scalability results for simple-mg on SGI-Altix.

SGI	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	9.6	5.5	2.6	5.4
2-2-2	6.0	3.6	1.4	3.0
4-4-4	1.3	2.9	0.8	1.8
8-8-8	0.7	0.6	0.3	0.7
8-1-8	1.0	1.0	1.6	2.5

Table A.4: Scalability results for simple-mg on SGI-ORIGIN.

AMD	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	4.6	4.6	1.9	4.1
2-2-2	2.3	2.5	0.9	2.0
4-4-4	1.0	3.2	0.9	1.2
4-1-4	1.4	1.1	2.4	2.8

Table A.5: Scalability results on AMD Athlon-Cluster.

AMD	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	1.5	1.6	0.6	1.1
4-4-4	1.1	1.0	0.2	0.3
8-8-8	0.4	0.5	0.1	0.2
8-1-8	0.7	0.6	0.6	0.7

Table A.6: Scalability results on AMD Opteron-Cluster.

In general the elapsed times are in the order of seconds for the simple-mg. Nevertheless scalability of OASIS4 PSMILe and Transformer can be demonstrated by comparing the 'enddef' or 'ping-pong' times for configurations 1-1-1, 2-2-2, 4-4-4, and 8-8-8 (when available). This time decreases on all platforms with the number of processes used (the only exceptions is the 'enddef OCE' time for the AMD Athlon PC Cluster for 4-4-4 on table A.5).

At the end of each table, the numbers for the 4-1-4 or 8-1-8 configuration are also given. This number illustrated the necessity of having a parallel Tranformer; in fact, the ping-pong tests realised with only 1 process for the Transformer (4-1-4 or 8-1-8 configuration) show an elapse time which is up to 3 times larger than the ping-pong tests realised with 4 or 8 processes for the Transformer (4-4-4 or 8-8-8 configuration).

The parallelization of the OASIS4 gives therefore big advantages in case of expensive interpolations between component fields exchanged between highly parallel component models. The parallel neighbourhood search in the PSMILe library as well as the parallel Transformer reduce interpolation time as well as communication time.

Bibliography

[1] Ahrem, R. et al., 2003: MpCCI Mesh-based parallel Code Coupling Interface, Specification Version 2, Fraunhofer Institute for Algorithms and Scientific Computing. Sankt Augustin, Germany.

[2] Balaji, 2001: Parallel Numerical Kernels for Climate Models, ECMWF TeraComputing Workshop 2001, World Scientific Press, Reading.

[3] Buja, L. and T. Craig, 2002: Community Climate System Model CCSM 2.0.1 User Guide, National Center of Atmospheric Research, Boulder, CO.

[4] Eaton, B., Gregory, J., Drach, B., Taylor, K., and Hankin, S.PMEL, 2003: NetCDF Climate and Forecast (CF) Metadata Conventions, http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html

[5] Gropp, W., S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, 1998: MPI – The Complete Reference, Vol. 2 The MPI Extensions, MIT Press.

[6] Legutke, S. and V.Gayler, 2004: The PRISM Standard Compilation Environment, PRISM Report Series No 4.

[7] Li, J., W. Liao, A. Choudhary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher, M. Zingale, 2003: Parallel NetCDF: A High-performance Scientific IO Interface, Proceedings of the SC'03, Nov 15-21, Phoenix, Arizona, USA. http://www-unix.mcs.anl.gov/parallel-netcdf

[8] Rew, R. K. and G. P. Davis, 1997: Unidata's netCDF Interface for Data Access: Status and Plans, Thirteenth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology, Anaheim, California, American Meteorology Society.

[9] Snir, M., S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, 1998: MPI - The Complete Reference, Vol. 1 The MPI Core, MIT Press.

[10] http://www.w3.org/XML/

[11] http://www.xmlsoft.org/