Supplement of Geosci. Model Dev., 16, 2833–2850, 2023
https://doi.org/10.5194/gmd-16-2833-2023-supplement

Geoscientific
Model Development
Open Access

Supplement of

C-Coupler3.0: an integrated coupler infrastructure for Earth system modelling

Li Liu et al.

Correspondence to: Li Liu (liuli-cess@tsinghua.edu.cn)

C-Coupler3 User Guide

Edited by:

Li Liu, Chao Sun, Xinzhu Yu, Hao Yu

23st, May, 2022

How to get assistance?

Assistance can be obtained as listed below

Electronic Mail Addresses

Name	**Affiliation**	**e-mail**
Li Liu	Department of Earth System Science, Tsinghua University	liuli-cess@tsinghua.edu.cn

Content

1 Introduction

The Community Coupler (C-Coupler) is a coupler family whose development was initiated in 2010 in China, primarily for developing coupled models for weather forecasting, climate simulation and prediction. It is targeted to serve various coupled models with flexibility, user-friendliness and sufficient coupling function supports. The first version C-Coupler1 was finished in 2014 and the second version C-Coupler2 was released 2018. C-Coupler is mainly programmed in C/C++, with a set of Fortran application programming interfaces (APIs) that can be called by component models. It has been parallelized by Message Passing Interfaces (MPI)[1]. After compilation, C-Coupler is a coupling library that can be further linked to the component models to serve model coupling and nesting. C-Coupler also supports file I/O using NetCDF. Compared to the previous version C-Coupler1, C-Coupler2 has a set of new features, including a common, flexible, and user-friendly coupling configuration interface that combines a set of application programming interfaces and a set of XML formatted configuration files, capability of coupling within one executable or the same subset of MPI (Message Passing Interface) processes, flexible and automatic coupling procedure generation for any subset of component models, dynamic 3-D coupling that enables convenient coupling of fields on 3-D grids with time-evolving vertical coordinate values, non-blocking data transfer, facilitation for model nesting, facilitation for increment coupling, adaptive restart capability and debugging capability.

C-Coupler3 is fully compatible with C-Coupler2. Model developers therefore can easily upgrade C-Coupler2 to C-Coupler3 in a coupled model without modifying model codes. Compared to C-Coupler2, C-Coupler3 achieves the following new features and advancements:

1) With a series of parallel optimization technologies, i.e., parallel triangulation of a horizontal grid (Yang et al., 2019), a distributed implementation of routing network generation for the data transfer (Yu et al., 2020), distributed management of horizontal grids and the corresponding parallel remapping weights calculation, and parallel input/output of remapping weights from/to a file, C-Coupler3 is able to handle the coupling under much finer resolutions (e.g., with more than 100 million horizontal grid cells) with fast coupling initialization.
2) C-Coupler3 can help to conveniently parallelize a component model with a common halo-exchange library. This library can support various horizontal grids and various halo regions, and enables to flexibly use asynchronous MPI communications for better parallel performance of the model.
3) C-Coupler3 enables a model to conveniently integrate and then use a software module, e.g., a flux algorithm, a parameterization scheme, a data assimilation method, etc., with a new framework (called common module integration framework hereafter). This framework can automatically and efficiently handle argument passing among a model and a module, even when they use different data structures or different grids. For example, a 3-D parameterization scheme can be easily integrated into a single-column data structure based physical package.
4) C-Coupler3 enables a model to conveniently integrate and then use a software module, e.g., a flux algorithm, a parameterization scheme, a data assimilation method, etc., with a new framework (called common module integration framework hereafter). This framework can automatically and

[1] Please note that C-Coupler3 has not been parallelized with OpenMP. If a component model has been parallelized with OpenMP, please make sure that only one thread of each MPI process calls a C-Coupler3 API at each time. Errors will happen if multiple threads of a process call a C-Coupler3 API at the same time.

efficiently handle argument passing among a model and a module, even when they use different data structures or different grids. For example, a 3-D parameterization scheme can be easily integrated into a single-column data structure based physical package.

5) C-Coupler3 includes a common framework (Sun et al., 2021) for conveniently developing a weakly coupled ensemble data assimilation (DA) system. This framework provides online data exchanges between a model ensemble and a DA method, for better parallel performance.
6) C-Coupler3 includes a common framework (Yu et al., 2022) for flexibly inputting and outputting fields in parallel. This framework has already been used by C-Coupler3 for improving the I/O of restart fields. Models can also benefit from it.

1.1 Step-by-step use of C-Coupler3

Users should take the following steps to use C-Coupler3 for model coupling, model nesting, model parallelization, module integration, data assimilation, I/O optimization, etc.

1) Obtain the C-Coupler3 source code.
2) Identify the model coupling resources (i.e., component models, model time and timers for each component model, model grids, parallel decompositions, halo regions, coupling field instances, coupling interfaces, I/O interfaces, external modules, data assimilation algorithms, etc.) through calling C-Coupler3 APIs (please refer to Section 2 for details).
3) Specify coupling configurations between component models based on the XML formatted configuration files and auxiliary data files (please refer to Section 3 for details). Create a working directory ("CCPL_dir") for C-Coupler3 and put these files under a sub directory ("CCPL_dir/config") of the working directory (Section 3).
4) Compile C-Coupler3 and the component models, and then run the coupled model.
5) Read the log files of C-Coupler3 when an error happens or when wanted.

1.2 C-Coupler3 sources

The C-Coupler3 directory structure is as follows:

- c_coupler/src	Source code of C-Coupler3. It includes several subdirectories: CoR, Data_MGT, Driver, Parallel_MGT, Runtime_MGT, PatCC, Utils and XML. The code under the subdirectory XML is used for parsing and writing XML files. It is the external code mainly developed by Lee Thomason (www.grinninglizard.com). The remaining code was developed by us.
/build	It includes a Makefile for compiling the C-Coupler3 source code into a library.
/doc	C-Coupler3 user guide.
/examples	Environment to compile, run and use different toy coupled models.

1.3 Licenses and copyrights

Copyright 2022 Tsinghua University. This software and ancillary information are called C-Coupler3. Access and use of this software shall impose the following obligations and understandings on the user. The user can be granted the right, without any fee or cost, to use, copy, modify, alter, enhance and distribute this software, and any derivative works thereof, and its supporting documentation for any purpose whatsoever, except any commercial activities (such as commercial usage, commercial sales, commercial services, etc.), provided that the following conditions are met:

1) This entire notice appears in all copies of the software, derivative works and supporting documentation.
2) If you modify the software and give a copy or copies of those modifications to others, either in source code or binary form, the software must carry notices stating that you changed specified portions of the Software.

This software is provided "as is" and any express or implied warranties, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event will Tsinghua University or any contributors to the software be liable for any damages, including direct, incidental, special, or consequential damages resulting from exercise of this license agreement or the use of the software. This material is made available with the understanding that Tsinghua University is not obligated to provide (and will not provide) the user with any support, consulting, training, or assistance of any kind with regard to the use, operation and performance of this software, nor to provide the user with any updates, revisions, new versions, or "bug fixes".

The licenses and copyrights of C-Coupler3 does not include external XML code mainly developed by Lee Thomason (www.grinninglizard.com). Please refer to the corresponding declarations in the external XML code files.

2 C-Coupler3 APIs

C-Coupler3 works as a library for model coupling. It provides a number of Fortran APIs that enable a component model to flexibly describe various coupling configurations. To enhance the software debugging capability regarding to C-Coupler3, the last parameter in each API is an optional input string named “annotation”. When “annotation” is given, C-Coupler3 will include it into an error report, which will enable users to easily locate the model code corresponding to an error report.

The C-Coupler3 APIs can be classified into the following types:

1) APIs for component model management
2) APIs for model time management
3) APIs for grid management
4) APIs for parallel decomposition management
5) APIs for coupling field management
6) APIs for coupling interface management
7) APIs for adaptive restart management
8) APIs for the common I/O framework
9) APIs of halo exchanges for model parallelization
10) APIs for the common module integration framework
11) APIs for the common data assimilation framework
12) APIs for parallel debugging

2.1 Module to use in the code

To use the C-Coupler3 library, a user needs to add in his Fortran code:

- **USE c_coupler_interface_mod**

2.2 APIs for component model management

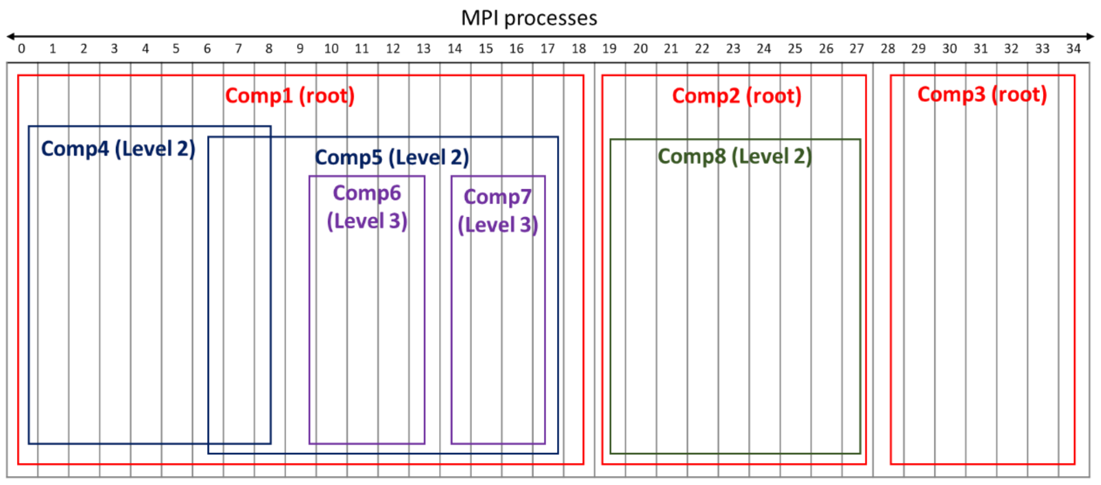


Figure 1 An example of MPI process layout among component models (comp1 ~ comp8).

To achieve model coupling between the component models running on non-overlapping, partially overlapping, or overlapping subsets of MPI processes, C-Coupler allows a component model to run on any subset of MPI processes. Therefore, C-Coupler can support almost any kind of MPI process layout among component models. Figure 1 shows an example of a complex MPI process layout: *comp1*, *comp2* and *comp3* do not share any MPI process; *comp4* runs on a proper subset of the MPI processes of *comp1*; *comp8* run on all MPI processes of *comp2*; *comp4* and *comp5* partially share some MPI processes. Moreover, there are relationships between the component models in Figure 1: *comp1* is the parent of *comp4* and *comp5*; *comp5* is the parent of *comp6* and *comp7*; *comp2* is the parent of *comp8*. In C-Coupler, a component model must cover all MPI processes of its children (for example, *comp1* in Figure 1 includes all processes of *comp4* and *comp5*). A component model that does not have the parent is called a root component model (for example, *comp1*, *comp2* and *comp3* in Figure 1 are root component models). There is a constraint regarding to root component models: each MPI process must belong to a unique root component model (for example, each process in Figure 1 only belongs to one of *comp1*, *comp2* and *comp3*), which means that all root component models cover all MPI processes while do not share any MPI process with each other. This constraint seems contradictory to the target of supporting shared MPI processes among component models, and may make C-Coupler unable to support some MPI process layouts. For example, given that a component model consists of two component models that run on partially overlapping subsets of MPI processes, the both component models cannot be root component models. To support this kind of MPI process layouts, the coupled model can be registered as a root component model of C-Coupler and its component models can be further registered as the children of the root component model.

Table 1 APIs for component model management

No.	API	Brief description
1	CCPL_register_component	Register a component model to C-Coupler
2	CCPL_get_component_id	Get the ID of a component model corresponding to the given model name
3	CCPL_get_current_process_id_in_component	Get the ID of the current MPI process in the given component model
4	CCPL_get_component_process_global_id	Get the global ID of a process in the given component model
5	CCPL_get_num_process_in_component	Get the total number of processes in the MPI communicator of the given component model
6	CCPL_get_local_comp_full_name	Get the full name of the given component model
7	CCPL_is_current_process_in_component	Check whether the current MPI process is in the given component model
8	CCPL_end_coupling_configuration	Finalize the stage of coupling configuration of a given component model
9	CCPL_finalize	Finalize the model coupling by C-Coupler
10	CCPL_is_comp_type_coupled	Check whether at least one component model with the given model type has already been registered to C-Coupler

The APIs for component model management are listed in Table 1. The API "*CCPL_register_component*" is responsible for registering a component model to C-Coupler. C-Coupler only serves the component models that have been registered to it (a component model whose model coupling is fully served by other couplers but not C-Coupler can be not registered to C-Coupler). The arguments of this API include the ID of the parent component model, model name, model type, MPI communicator, etc. Any component model except a root component model must have a parent. C-Coupler will allocate an ID and generate a unique full name for each component model. The full name is formatted as "*parent_full_name@model_name*", where "*model_name*" means the name of the current component model and "*parent_full_name*" means the full name of the parent component model ("*parent_full_name*" of a root component model corresponding to an empty string). A component model is either active or pseudo (inactive), which is specified through the model type. A pseudo component model can be the parent of some component models while its name will not be included in the full name of any component model. Moreover, coupling configurations cannot be further specified to a pseudo component model. Table 2 lists out the model types that are currently supported by C-Coupler. Please note that the model types of "active_coupled_system" and "pseudo_coupled_system" indicate that an existing coupled model can be registered as a component model of C-Coupler. This API can create the MPI communicator of the component model when required. It will start the stage of coupling configuration of the component model, while the API "*CCPL_end_coupling_configuration*" will finalize the stage of coupling configuration. A component model can successfully call "*CCPL_end_coupling_configuration*" only when all its children component models have already called this API.

Table 2 Model types that are currently supported by C-Coupler

Model type	Description	Remark
Cpl	Coupler	Active component model
Atm	Atmosphere model	Active component model
Glc	Glacier model	Active component model
atm_chem	Atmospheric chemistry model	Active component model
Ocn	Ocean model	Active component model
Lnd	Land surface model	Active component model
sea_ice	Sea ice model	Active component model
Wave	Wave model	Active component model
Roff	Runoff model	Active component model
active_coupled_system	Coupled model that consists of a set of component models	Active component model
pseudo_coupled_system	Coupled model that consists of a set of component models	Pseudo component model

2.2.1 CCPL_register_component

- **integer FUNCTION CCPL_register_component(parent_id, comp_name, comp_type, comp_comm, considered_in_ancestor_coupling_gen, change_dir, annotation)**
  - return value [INTEGER; OUT]: The ID of the new component model
  - parent_id [INTEGER; IN]: The ID of the parent component model. When the new

component model is a root component model that does not have the parent, “parent_id” should be -1.

- comp_name [CHARACTER; IN]: The name of the new component model. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, ‘0’~‘9’ or ‘_’. “comp_name” should not include a prefix of “ROOT”, “DATA_MODEL”, “DATA_INST” or “ALGORITHM_MODEL”.
- comp_type [CHARACTER; IN]: The type of the new component mode. The model types that are currently supported by C-Coupler are shown in Table 2.
- comp_comm [INTEGER; INOUT]: The MPI communicator corresponding to the new component model. “comp_comm” can be specified as a C-Coupler pre-defined variable *CCPL_NULL_COMM*, which means the MPI communicator corresponding to the new component model is unknown and then will be created by C-Coupler and returned through “comp_comm”. “comp_comm” can also be a known MPI communicator that has already been created beyond C-Coupler. When “comp_comm” is not *CCPL_NULL_COMM*, users must guarantee that “comp_comm” is a legal MPI communicator.
- considered_in_ancestor_coupling_gen [LOGICAL, OPTIONAL; IN]: It specifies whether to consider the new component model in the family coupling procedure generation of its parent component model (please refer to Section 2.8). Its default value is *true*, which means to consider the new component model in the family coupling procedure generation of its parent component model.
- change_dir [LOGICAL, OPTIONAL; IN]: It specifies whether to change the working directory of the current MPI process to the corresponding working directory of the C-Coupler platform. Its default value is *false*, which means not to change the working directory. The working directory can be changed only when registering a root component model. In other words, it is meaningless to specify the value of “change_dir” and the working directory will not be changed when registering a non-root component model.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API registers a new component model to C-Coupler and returns its ID when the registration succeeds. For each process, the first component model to be registered must be the root component and only one root component can be registered. If “comp_comm” is *CCPL_NULL_COMM*, all MPI processes in the parent component model (corresponding to “parent_id”) must call this API at the same time (when “parent_id” is -1, all MPI processes in the communicator MPI_COMM_WORLD must call this API at the same time), and “comp_name” is the key for a component model (the MPI processes with the same “comp_name” will be classified into the same component model). If “comp_comm” is a valid MPI communicator, all MPI processes in this MPI communicator must call this API at the same time with the same “comp_name” and “comp_type”. Any two component models on the same MPI process cannot share the same “comp_name”. All component models registered by the same MPI process cannot share the same “comp_name”. This API will start the stage of coupling configuration of the corresponding component model.

2.2.2 CCPL_get_component_id

- **integer FUNCTION CCPL_get_component_id(comp_name, annotation)**
  - Return value [INTEGER; OUT]: The ID of the corresponding component model. A return value of -*1* means that no component model with the given "comp_name" is found.
  - comp_name [CHARACTER; IN]: The name of a component model. It has a maximum length of 80 characters.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the ID of a component model that has been registered in the current MPI process, according to the given model name.

2.2.3 CCPL_get_current_process_id_in_component

- **integer FUNCTION CCPL_get_current_process_id_in_component(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The ID of the current MPI process in the MPI communicator of the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the ID of the current MPI process in the MPI communicator of the given component model that should have already been registered to C-Coupler in the current MPI process.

2.2.4 CCPL_get_num_process_in_component

- **integer FUNCTION CCPL_get_num_process_in_component(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The total number of MPI processes in the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the total number of processes in the given component model that should have already been registered to C-Coupler in the current MPI process.

2.2.5 CCPL_get_component_process_global_id

- **integer FUNCTION CCPL_get_component_process_global_id(comp_id, local_proc_id,**

annotation)
- Return value [INTEGER; OUT]: The global ID of the given process in the given component model.
- comp_id [INTEGER; IN]: The ID of the given component model
- local_proc_id [INTEGER; IN]: The local ID of the given process in the given component model. It should be no smaller than 0 and smaller than the total number of processes in the given component model.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API returns the global ID of a given MPI process in the given component model that should have already been registered to C-Coupler in the current MPI process.

2.2.6 CCPL_get_local_comp_full_name

• **SUBROUTINE CCPL_get_local_comp_full_name(comp_id, comp_full_name, annotation)**
- comp_id [INTEGER; IN]: The ID of the given component model
- comp_full_name [CHARACTER; OUT]: The full name of the given component model. This string should have sufficient length, for example with a length of 512 characters.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API gets the full name of the given component model that should have already been registered to C-Coupler in the current MPI process.

2.2.7 CCPL_is_current_process_in_component

• **logical FUNCTION CCPL_is_current_process_in_component(comp_full_name, annotation)**
- Return value [LOGICAL; OUT]: If the current MPI process is in the given component model, “true” will be returned; otherwise “false” will be returned.
- comp_full_name [CHARACTER; IN]: The full name of the given component model. It has a maximum length of 512 characters.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API checks whether the current MPI process is in the given component model that is specified through the full name.

2.2.8 CCPL_is_comp_type_coupled

- **logical FUNCTION CCPL_is_comp_type_coupled(comp_id, comp_type, annotation)**
  - Return value [LOGICAL; OUT]: If any component model whose type is "*comp_type*" has been registered to C-Coupler, "true" will be returned; otherwise "false" will be returned.
  - comp_id [INTEGER; IN]: The ID of a given component model
  - comp_type [CHARACTER; IN]: a type of component models. It must be a model type supported by C-Coupler.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API checks whether at least one component model whose type is "*comp_type*" has already been registered to C-Coupler, for the given component model. All MPI processes of the given component model are required to call this API at the same time, with the same input parameter "*comp_type*".

2.2.9 CCPL_end_coupling_configuration

- **SUBROUTINE CCPL_end_coupling_configuration(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API finalizes the stage of coupling configuration of the given component model that should have already been registered to C-Coupler in the current MPI process. A component model can successfully call this API only when all its children component models have already called this API. After calling this API, the corresponding component model cannot further register children component models, timers, grids, parallel decompositions, coupling field instances, coupling interfaces, etc. All MPI processes in the communicator MPI_COMM_WORLD are required to call this API at the same time. When the given component model is a root component model and the input parameter "considered_in_ancestor_coupling_gen" when registering the given component model is not specified or has been set to *true*, the overall coupling procedure generation with the given component model will be conducted automatically. Please note that, this API can be called at most one time.

2.2.10 CCPL_finalize

- **SUBROUTINE CCPL_finalize(to_finalize_MPI, annotation)**
  - to_finalize_MPI [LOGICAL; IN]: when "to_finalize_MPI" is set to "true" and MPI has not been finalized, MPI will be finalized. C-Coupler will not finalize MPI when "to_finalize_MPI" is set to "false".
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API releases all data structures of C-Coupler and then finalize C-Coupler. It can also finalize MPI at the same time when “to_finalize_MPI” has not been set to “false”. All MPI processes in the communicator MPI_COMM_WORLD must call this API at the same time, with consistent parameters. For each MPI process, “CCPL_finalize” can be called only once. We propose to call “CCPL_finalize” (“to_finalize_MPI” should be set to “*true*”) before calling MPI_Finalize in the original model code (if have). Errors may happen when calling MPI_Finalize before calling “CCPL_finalize”.

2.3 APIs for model time management

Table 3 lists out the APIs for time management that enable C-Coupler to manage the model time information for each active component model. Detailed time information of a component model can also be accessed through C-Coupler and thus a component model can employ C-Coupler for its model time management. For a component model that has its own model time management, it should be guaranteed that the model time is constantly consistent between the component model and C-Coupler. The API “*CCPL_check_current_time*” can be used to check such consistency. An active component model can have a unique time manager that is not activated until the unique time step has been set through the API “*CCPL_set_normal_time_step*”. After a time manager is activated, users can access detailed information of the model time, define timers, advance the model time and use timers to control model coupling.

Besides managing the time information of each component model, the time managers of C-Coupler also share some identical global information of a simulation run (specified in the XML file “CCPL_dir/config/all/env_run.xml” (p)), including the case name, case description, run type (“initial”, “continue”, “branch” and “hybrid” are the four run types currently supported in C-Coupler), start time, stop time (the simulation run can be set to endless), and the frequency to write restart files. Please note that C-Coupler requires all component models in a coupled system to use the same case name, case description, run type, start time, stop time and restart writing frequency.

Currently, C-Coupler only supports the model years from 0 to 9999. When users want C-Coupler to support the model years beyond this range, please contact the authors.

Table 3 APIs for time management.

No.	API	Brief description
1	CCPL_set_normal_time_step	Set the unique time step of the given component model.
2	CCPL_get_normal_time_step	Get the time step of the given component model, which is set by the API “CCPL_set_normal_time_step”.
3	CCPL_advance_time	Advance the model time of the given component model by a time step.
4	CCPL_get_number_of_current_step	Get the number of current time step of the given component model.
5	CCPL_get_number_of_total_steps	Get the total number of time steps during the whole

		simulation of the given component model.
6	CCPL_get_current_date	Get the current date of the given component model.
7	CCPL_get_current_year	Get the current year of the given component model.
8	CCPL_get_current_num_days_in_year	Get the current number of days in the current year of the given component model.
9	CCPL_get_current_second	Get the current second of the given component model.
10	CCPL_get_start_time	Get the start time of model run of the given component model.
11	CCPL_get_stop_time	Get the stop time of model run of the given component model.
12	CCPL_get_previous_time	Gets the time of the previous time step of the given component model.
13	CCPL_get_current_time	Get the current time of the given component model.
14	CCPL_is_first_step	Check whether the current time step is the first step in a model run.
15	CCPL_is_first_restart_step	Check whether the current time step is the first step after restarting the simulation run (first restart step). Please note that the first restart step may not be the first step of the model run.
16	CCPL_get_num_elapsed_days_from_start	Get the number of elapsed days from the start date to the current date of the given component model.
17	CCPL_get_num_elapsed_days_from_reference	Get the number of elapsed days from the reference date to the current date of the given component model.
18	CCPL_is_end_current_day	Check whether the current time is the end of the current day for the given component model.
19	CCPL_is_end_current_month	Check whether the current time is the end of the current month for the given component model.
20	CCPL_get_current_calendar_time	Get the current calendar time of the given component model.
21	CCPL_check_current_time	Check the consistency of model time between the given component model and C-Coupler
22	CCPL_is_model_run_ended	Check whether the simulation run of the given component model has reached the stop time.
23	CCPL_define_single_timer	Define a single timer that is a periodic timer for the given component model.
24	CCPL_is_timer_on	Check whether the given timer is on at the current time step of the corresponding component model.
25	CCPL_is_last_step_of_model_run	Check whether the current time step is the last time step of model run of the given component model.
26	CCPL_reset_current_time_to_start_time	Reset the current time of the given component model to the start time of the model run.

2.3.1 CCPL_set_normal_time_step

- **SUBROUTINE CCPL_set_normal_time_step(comp_id, time_step_in_second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - time_step_in_second [INTEGER; IN]: The unique time step (number of seconds) of the given component model. It must be an integer value larger than 0.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API sets the unique time step (number of seconds) of the given component model. Currently, C-Coupler requires the total number of seconds of the whole simulation run to be an integral multiple of "time_step_in_second", and requires the frequency (in seconds) of writing restart data file to be an integral multiple of "time_step_in_second". All MPI processes of the given component model are required to call this API at the same time with the same "time_step_in_second". A component model can call this API only one time. In other word, a component model can have at most one time step. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.3.2 CCPL_get_normal_time_step

- **integer FUNCTION CCPL_get_normal_time_step(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The time step (number of seconds) of the given component model. When the time step of the corresponding component model has not been set, the return value will be -*1*.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the time step of the given component model.

2.3.3 CCPL_advance_time

- **SUBROUTINE CCPL_advance_time(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API advances the time of the corresponding component model by one time step. All MPI processes of the component model are required to call this API at the same time.

2.3.4 CCPL_get_number_of_current_step

- **integer FUNCTION CCPL_get_number_of_current_step(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The number of the current time step in the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the current time step in the component model corresponding to “comp_id”. The number of the first time step in an initial run, or hybrid run is 0. A restart run or a branch run shares the same number of time step with the corresponding initial run. Therefore, the number of the first time step of a restart run or a branch run may not be 0.

2.3.5 CCPL_get_number_of_total_steps

- **integer FUNCTION CCPL_get_number_of_total_steps(comp_id, annotation)**
  - return value [INTEGER; OUT]: The total number of time steps from the start to the end of the simulation run of the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the total number of time steps from the start to the end of the simulation run of the given component model. When the simulation run has been set to be endless, “-1” will be returned.

2.3.6 CCPL_get_current_date

- **integer FUNCTION CCPL_get_current_date(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The current date (YYYYMMDD) of the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the current date of the given component model.

2.3.7 CCPL_get_current_year

- **integer FUNCTION CCPL_get_current_year(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The current year (YYYY) of the given component

model.
- comp_id [INTEGER; IN]: The ID of the given component model
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the current year of the given component model.

2.3.8 CCPL_get_current_num_days_in_year

- **integer FUNCTION CCPL_get_current_num_days_in_year(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The current number of days (1~365 or 1~366) in the current year of the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the current number of days (1~365 or 1~366) in the current year of the given component model.

2.3.9 CCPL_get_current_second

- **integer FUNCTION CCPL_get_current_second(comp_id, annotation)**
  - Return value [INTEGER; OUT]: The current second number (<86400) in the current day of the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the current second number in the current day of the given component model.

2.3.10 CCPL_get_start_time

- **SUBROUTINE CCPL_get_start_time(comp_id, year, month, day, second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - year [INTEGER; OUT]: The year (YYYY) of the start time of the given component model.
  - month [INTEGER; OUT]: The month (MM) of the start time of the given component model.
  - day [INTEGER; OUT]: The day (DD) of the start time of the given component model.
  - second [INTEGER; OUT]: The second number (<86400) of the start time of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the

corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API gets the start time (year, month, day and second) of the model run of the given component model.

2.3.11 CCPL_get_stop_time

- **SUBROUTINE CCPL_get_stop_time(comp_id, year, month, day, second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - year [INTEGER; OUT]: The year (YYYY) of the stop time of the given component model
  - month [INTEGER; OUT]: The month (MM) of the stop time of the given component model
  - day [INTEGER; OUT]: The day (DD) of the stop time of the given component model.
  - second [INTEGER; OUT]: The number of second (<86400) of the stop time of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API gets the stop time (year, month, day and second) of the model run of the given component model. When the simulation run has been set to be endless, "-1" will be returned through each output parameter.

2.3.12 CCPL_get_previous_time

- **SUBROUTINE CCPL_get_previous_time(comp_id, year, month, day, second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - year [INTEGER; OUT]: The year (YYYY) at the previous time step of the given component model
  - month [INTEGER; OUT]: The month (MM) at the previous time step of the given component model
  - day [INTEGER; OUT]: The day (DD) at the previous time step of the given component model.
  - second [INTEGER; OUT]: The second number (<86400) at the previous time step of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API gets the time at the previous time step of the given component model. The previous time of the start time is the start time itself.

2.3.13 CCPL_get_current_time

- **SUBROUTINE CCPL_get_current_time(comp_id, year, month, day, second, shift_second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - year [INTEGER; OUT]: The year (YYYY) of the current time of the given component model
  - month [INTEGER; OUT]: The month (MM) of the current time of the given component model
  - day [INTEGER; OUT]: The day (DD) of the current time of the given component model.
  - second [INTEGER; OUT]: The second number (<86400) of the current time of the given component model.
  - shift_second [INTEGER, OPTIONAL; IN]: An additional number of seconds (≥0) from the current time when getting the current time. For example, when "shift_second" is 60, the time at 60 seconds after the current time will be returned.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API gets the current time of the given component model. An additional number of seconds can be added when getting the current time.

2.3.14 CCPL_is_first_step

- **logical FUNCTION CCPL_is_first_step(comp_id, annotation)**
  - return value [LOGICAL; OUT]: if the current time step is the first step of a simulation run, the return value is *true*; otherwise, the return value is *false*.
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API checks whether the current time step is the first time step of a simulation run. Please note that the first step after restarting the simulation run (first restart step) may not be the first time step.

2.3.15 CCPL_is_first_restart_step

- **logical FUNCTION CCPL_is_first_restart_step(comp_id, annotation)**
  - Return value [LOGICAL; OUT]: if the current time step is the first step after restarting the simulation run (first restart step), the return value is true; otherwise, the return value is false.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API checks whether the current time step is the first step after restarting the simulation run (first restart step). Please note that the first restart step may not be the first step.

2.3.16 CCPL_get_num_elapsed_days_from_start

- **SUBROUTINE CCPL_get_num_elapsed_days_from_start(comp_id, num_days, current_second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - num_days [INTEGER; OUT]: The number of elapsed days from the start date to the current date of the given component model.
  - current_second [INTEGER; OUT]: The second number (<86400) of the current time of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API gets the number of elapsed days from the start date to the current date and gets the current second of the given component model.

2.3.17 CCPL_get_num_elapsed_days_from_reference

- **SUBROUTINE CCPL_get_num_elapsed_days_from_reference(comp_id, num_days, current_second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - num_days [INTEGER; OUT]: The number of elapsed days from the reference date to the current date of the given component model.
  - current_second [INTEGER; OUT]: The second in a day of the current time of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API gets the number of elapsed days from the reference date to the current date and gets the current second of the corresponding component model.

2.3.18 CCPL_is_end_current_day

- **logical FUNCTION CCPL_is_end_current_day(comp_id, annotation)**
  - return value [LOGICAL; OUT]: if the current time is the end of the current day, the return value is *true*; otherwise, the return value is *false*.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API checks whether the current time is the end of the current day (whether the current second is 0) for the given component model.

2.3.19 CCPL_is_end_current_month

- **logical FUNCTION CCPL_is_end_current_month(comp_id, annotation)**
  - Return value [LOGICAL; OUT]: if the current time is the end of the current month of the given component model, the return value is *true*; otherwise, the return value is *false*.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API checks whether the current time is the end of the current month (whether the current day is 1 and the current second is 0) for the given component model.

2.3.20 CCPL_get_current_calendar_time

- **SUBROUTINE CCPL_get_current_calendar_time(comp_id, cal_time, shift_second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - cal_time [REAL; OUT]: The calendar time of the current time. It is a non-negative floating-point value no more than the number of days in a year.
  - shift_second [INTEGER, OPTIONAL; IN]: An additional number of seconds (≥0) from the current time when calculating the calendar time.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API gets the current calendar time of the given component model. A positive number of seconds can be added as a shift when calculating the calendar time.

2.3.21 CCPL_check_current_time

- **SUBROUTINE CCPL_check_current_time(comp_id, date, second, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - date [INTEGER; in]: the date (YYYYMMDD) provide by the given component model.
  - second [INTEGER, OPTIONAL; IN]: the number of second provide by the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API checks whether the current time of the given component model is the same as the provided date and second. An error will be reported and the model run will be ended if the checking fails.

2.3.22 CCPL_is_model_run_ended

- **logical FUNCTION CCPL_is_model_run_ended(comp_id, annotation)**
  - return value [LOGICAL; OUT]: if the model run of the given component model has been ended, "true" will be returned; otherwise "false" will be returned.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API checks whether the simulation run of the given component model has reached the stop time.

2.3.23 CCPL_is_last_step_of_model_run

- **logical FUNCTION CCPL_is_last_step_of_model_run(comp_id, annotation)**
  - return value [LOGICAL; OUT]: if the current time step is the last time step of the model run of the given component model, "true" will be returned; otherwise "false" will be returned.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API checks whether the current time step is the last time step of the model run of the given component model.

2.3.24 CCPL_define_single_timer

- **integer FUNCTION CCPL_define_single_timer(comp_id, period_unit, period_count, local_lag_count, remote_lag_count, annotation)**
  - Return value [INTEGER; OUT]: The ID of the new timer.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - period_unit [CHARACTER; IN]: The unit for specifying the period of the single timer. The unit must be "step" ("steps", "nstep" or "nsteps"), "second" ("seconds", "nsecond" or "nseconds"), "day" ("days", "nday" or "ndays"), "month" ("months", "nmonth" or "nmonths"), or "year" ("years", "nyear" or "nyears").
  - period_count [INTEGER; IN]: A count (>0) corresponding to the unit for specifying the period of the single timer.
  - local_lag_count [INTEGER; IN]: A count corresponding to the period unit. It is used to specify a lag (can be viewed as a time offset from the start time) which will impact when

the single timer is on.

- remote_lag_count [INTEGER, OPTIONAL; IN]: A count corresponding to the period unit. It can be used to specify a lag for a coupling connection between two component models or intra one component model. Its default value is 0, which means no lag. Please note that the lag for a coupling connection is determined by the timer of the receiver component model. The lag corresponding to a coupling connection can be viewed as a time difference from the receiver component model to the sender component model, which can control the time sequence between two component models. For example, given a lag of 1 hour, the coupling fields produced by the sender component model at the sender's 0th hour will be obtained by the receiver component model at the receiver's 1st hour; given a lag of -1 hour, the coupling fields produced by the sender component model at the sender's 1st hour will be obtained by the receiver component model at the receiver's 0th hour. Thus, users can flexibly achieve concurrent run or sequential run between component models. Please note that, wrong setting of "remote_lag_count" may introduce deadlocks between component models.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API defines a single timer that is a periodic timer for the given component model. "period_unit" and "period_count" are used to specify the period of the single timer. In default, the single timer will be on at the start time of simulation. When users want to change the model time when the single timer is on, users can set a lag count (with positive or negative value) through the parameter "local_lag_count". The period of the single timer should be consistent with the time step size of the component model, which means the period in should be an integer multiple of the time step size. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

For example, give a single timer of <period_unit="steps", period_count="5", local_lag_count="2">, it will be on at the 2nd, 7th (5**i*+2, where *i* is a non-negative integer) time step of the given component model.

2.3.25 CCPL_is_timer_on

- **logical FUNCTION CCPL_is_timer_on(timer_id, annotation)**
  - Return value [LOGICAL; OUT]: If the given timer is on at the current time step of the corresponding component model, *true* will be returned; otherwise *false* will be returned.
  - timer_id [INTEGER: IN]: the ID of the given timer.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API checks whether the given timer is on at the current time step of the corresponding component model.

2.3.26 CCPL_reset_current_time_to_start_time

- **SUBROUTINE CCPL_reset_current_time_to_start_time(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API resets the current time of the given component model to the start time of the model simulation. All MPI processes of the given component model are required to call this API at the same time. This API can be called for a component model only when the component model has not executed any coupling interface.

Table 4 APIs for grid management.

No.	API	Brief description
1	CCPL_register_H2D_grid_via_global_data	Register a horizontal grid using global grid data
2	CCPL_register_H2D_grid_via_local_data	Register a horizontal grid using local grid data
3	CCPL_register_H2D_grid_via_file	Register a horizontal grid using a data file
4	CCPL_register_H2D_grid_from_another_component	Register a horizontal grid based on another component
5	CCPL_register_H2D_grid_without_data	Register a horizontal grid without grid data. Such a grid cannot be used in data interpolation
6	CCPL_register_V1D_Z_grid_via_model_data	Register a vertical Z grid using model data
7	CCPL_register_V1D_SIGMA_grid_via_model_data	Register a vertical SIGMA grid using model data
8	CCPL_register_V1D_HYBRID_grid_via_model_data	Register a vertical HYBRID grid using model data
9	CCPL_register_V1D_grid_without_data	Register a vertical grid without grid data. The vertical coordinate values of a 3-D grid with this vertical grid can be further set via the API CCPL_set_3D_grid_external_surface_field
10	CCPL_register_1D_normal_grid_without_data	Register a 1-D grid without grid data. Such a 1-D grid cannot be used in data interpolation
11	CCPL_register_MD_grid_via_multi_grids	Register a grid using multiple registered grids
12	CCPL_register_mid_point_grid	Register of the middle-point grid of a 3-D interface-level grid
13	CCPL_set_3D_grid_variable_surface_field	Set the dynamic surface field of a 3-D grid
14	CCPL_set_3D_grid_constant_surface	Set the static surface field of a 3-D grid

	_field	
15	CCPL_set_3D_grid_external_surface_field	Declare that the surface field of a 3-D grid is external
16	CCPL_set_3D_grid_3D_vertical_coord_field	Set the 3-D vertical coordinate values of a 3-D grid via a 3-D field instance
17	CCPL_get_grid_size	Get the global size of a grid
18	CCPL_get_grid_id	Get the ID of a grid
19	CCPL_get_H2D_grid_data	Get grid data of a horizontal grid

2.4 APIs for grid management

Each grid managed by C-Coupler belongs to a unique active component model. For a grid that is shared by multiple component models, it should be registered to each component model separately. The keyword for a grid can be expressed as <*ID of the component model*, *grid name*>. Therefore, different grids in the same component model cannot have the same grid name, while the grids in different component models can have the same grid name.

C-Coupler can manage various kinds of horizontal grids (2-D), vertical grids (1-D) with Z, SIGMA or HYBRID coordinates, and time grids (1-D), and also can manage multi-dimension grids each of which consists of a horizontal grid, a vertical grid or a time grid. C-Coupler offers multiple choices for registering model grids. A component model can register a model grid to C-Coupler through model data or a grid data file, or from another component model.

For a 3-D grid that consists of a horizontal grid and a vertical grid with SIGMA or HYBRID coordinate, C-Coupler enables to set its unique surface field that is on the horizontal grid, in order for calculating the vertical coordinates at each horizontal grid point. The surface field of a 3-D grid can be static, dynamic and external. A static surface field means that its values are constant along with the time integration, so that the vertical coordinate values in the corresponding 3-D grid are constant. A dynamic surface field means that its values change along with the time integration, so that the vertical coordinate values in the corresponding 3-D grid are variable. An external surface field means that its values are determined by the surface field of another 3-D grid. C-Coupler also enables to set the 3-D vertical coordinate values of a 3-D grid via a 3-D field instance.

Most models are parallelized with MPI, where the model grids are decomposed into separate domains for parallel integration. Besides managing global model grids, C-Coupler also manages decomposed model grids at the same time. Model code can access the grid data in a global model grid or the grid data in decomposed model grid.

Table 4 lists out the APIs for grid management as well as their brief descriptions.

2.4.1 CCPL_register_H2D_grid_via_global_data

- **integer FUNCTION CCPL_register_H2D_grid_via_global_data(comp_id, grid_name, edge_type, coord_unit, cyclic_or_acyclic, dim_size1, dim_size2, min_lon, max_lon, min_lat, max_lat, center_lon, center_lat, mask, area, vertex_lon, vertex_lat, annotation)**
  - return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is

associated with.

- grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.
- edge_type [CHARACTER; IN]: The type of the edges of grid cells (“LON_LAT”， “XY”, “GREAT_ARC”, or “TriPolar”).
- coord_unit [CHARACTER; IN]: The unit of the coordinate values of the new grid. When the edge type is “LON_LAT”, “GREAT_ARC” or “TriPolar”, the unit must be “degrees” or “radians”.
- cyclic_or_acyclic [CHARACTER; IN]: Whether the longitude (X) dimension is cyclic or acyclic. The value must be “cyclic” or “acyclic”. When the “edge_type” is “XY”, the “cyclic_or_acyclic” must be “acyclic”.
- dim_size1 [INTEGER; IN]: The size of the longitude (X) dimension or the global grid. “dim_size1” must be larger than 3.
- dim_size2 [INTEGER; IN]: If “dim_size1” has been set to the size of the global grid, “dim_size2” must be set to 0; otherwise, “dim_size2” must be set to the size of latitude (Y) dimension. “dim_size2” must be 0 or larger than 3.
- min_lon [REAL; IN]: Minimum longitude value of the boundary of the domain of the horizontal grid. Its value can be a specific value between -*360.0* and *360.0* degrees (or between -*2PI* and *2PI*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate “min_lon” automatically. Please note that, “min_lon” calculated by C-Coupler may be incorrect.
- max_lon [REAL; IN]: Maximum longitude value of the boundary of the domain of the horizontal grid. Its value can be a specific value between -*360.0* and *360.0* degrees (or between -*2PI* and *2PI*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate “max_lon” automatically. Please note that, “max_lon” calculated by C-Coupler may be incorrect. “min_lon” and “max_lon” must be -*999999.0* or not -*999999.0* at the same time. When both “min_lon” and “max_lon” are not -*999999.0*, “min_lon” can be larger or smaller than “max_lon”.
- min_lat [REAL; IN]: Minimum latitude value of the boundary of the domain of the horizontal grid. Its value can be a specific value between -*90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate “min_lat” automatically. Please note that, “min_lat” calculated by C-Coupler may be incorrect. When “min_lat” has been set to -*90* degrees (or *–PI/2*), it indicates that the global grid covers the South Pole.
- max_lat [REAL; IN]: Maximum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between -*90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate “max_lat” automatically. Please note that, “max_lat” calculated by C-Coupler may be incorrect. When “max_lat” has been set to *90* degrees (or *PI/2*), it indicates that the global grid covers the North Pole. When both “min_lat” and “max_lat” are not -*999999.0*, “min_lat” must be smaller than “max_lat”.
- center_lon [REAL, DIMENSION|(:) or (:,:)|; IN]: The longitude (X) value of the center of each grid cell. If “dim_size2” is larger than 3, the array size of “center_lon” can be “dim_size1” or “dim_size1”*“dim_size2” (the grid size); otherwise, the array size of

"center_lon" must be "dim_size1".

- center_lat [REAL, DIMENSION|(:) or (:,:)|; IN]: The latitude (Y) value of the center of each grid cell. If "dim_size2" is larger than 3 and the array size of "center_lon" is "dim_size1", the array size of "center_lat" must be "dim_size2"; otherwise, "center_lat" must have the same array size with "center_lon". "center_lat" must have the same number of dimensions with "center_lon".
- mask [INTEGER, DIMENSION|(:) or (:,:)|, OPTIONAL; IN]: The mark to specify whether each grid cell is active or not. The "mask" value of *1* means being active while value of *0* means being inactive. The array size of "mask" must be the grid size. "mask" must have the same number of dimensions with "center_lon". When "mask" is not provided, it means all grid cells are active.
- area [REAL, DIMENSION|(:) or (:,:)|, OPTIONAL; IN]: The area of each grid cell. The array size of "area" must be the grid size. "area" must have the same number of dimensions with "center_lon". The unit of "area" should be consistent with "center_lon".
- vertex_lon [REAL, DIMENSION|(:,:) or (:,:,:)|, OPTIONAL; IN]: The longitude (X) values of the vertexes of each grid cell. Corresponding to "center_lon", "vertex_lon" must have an additional lowest dimension whose size is the maximum number of vertexes. *-999999.0* can be used to specify the missing values.
- vertex_lat [REAL, DIMENSION|(:,:) or (:,:,:)|, OPTIONAL; IN]: The latitude (Y) values of the vertexes of each grid cell. Corresponding to "center_lat", "vertex_lat" must have an additional lowest dimension whose size is the maximum number of vertexes. *-999999.0* can be used to specify the missing values. "vertex_lon" and "vertex_lat" must be provide at the same time and have the same size at the lowest dimension.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

- **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to "comp_id" using the global grid data provided by the component model and returns the ID of the horizontal grid when the registration succeeds. It targets to support any kind of horizontal grids such as unstructured grids and longitude-latitude grids. For an unstructured grid, users are advised to specify the grid size through "dim_size1" while setting "dim_size2" to 0, and make the array size of "center_lon" and "center_lat" the same as the grid size. For a longitude-latitude grid, users can treat it as an unstructured grid, and can also specify the size of the longitude dimension through "dim_size1" and specify the size of the latitude dimension through "dim_size2" (in such a case, the array size of "center_lon" and "center_lat" can be the same as the grid size, or the same as "dim_size1" and "dim_size2" respectively). "Mask", "area", "vertex_lon" and "vertex_lat" are optional parameters. "vertex_lon" and "vertex_lat" are required to be provided when this horizontal grid is involved in generating the remapping weights of the conservative remapping algorithms.

All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.2 CCPL_register_H2D_grid_via_local_data

- **integer FUNCTION CCPL_register_H2D_grid_via_local_data(comp_id, grid_name, edge_type, coord_unit, cyclic_or_acyclic, grid_size, num_local_cells, local_cells_global_index, min_lon, max_lon, min_lat, max_lat, center_lon, center_lat, mask, area, vertex_lon, vertex_lat, decomp_name, decomp_id, annotation)**
  - return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - edge_type [CHARACTER; IN]: The type of the edges of grid cells ("LON_LAT"， "XY", "GREAT_ARC", or "TriPolar").
  - coord_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. When the edge type is "LON_LAT", "GREAT_ARC" or "TriPolar", the unit must be "degrees" or "radians".
  - cyclic_or_acyclic [CHARACTER; IN]: whether the longitude (X) dimension is cyclic or acyclic. The value must be "cyclic" or "acyclic". When the "edge_type" is "XY", the "cyclic_or_acyclic" must be "acyclic".
  - grid_size [INTEGER; IN]: The size of the size of the global grid (must be larger than 3).
  - num_local_cells [INTEGER; IN]: The number of grid cells (≥0) corresponding to the local data of the current MPI process.
  - local_cells_global_index [INTEGER, DIMENSION(:); IN]: The global index of the grid cells corresponding to the local data of the current MPI process. The array size of "local_cells_global_index" cannot be smaller than "num_local_cells". Each value in "local_cells_global_index" must be between *1* and the "grid_size".
  - min_lon [REAL; IN]: Minimum longitude value of the boundary of the domain of the global grid. Its value can be a specific value between -*360.0* and *360.0* degrees (or between -*2PI* and *2PI*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate "min_lon" automatically. Please note that, "min_lon" calculated by C-Coupler may be incorrect.
  - max_lon [REAL; IN]: Maximum longitude value of the boundary of the domain of the global grid. Its value can be a specific value between -*360.0* and *360.0* degrees (or between -*2PI* and *2PI*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate "max_lon" automatically. Please note that, "max_lon" calculated by C-Coupler may be incorrect. "min_lon" and "max_lon" must be -*999999.0* or not -*999999.0* at the same time. When both "min_lon" and "max_lon" are not -*999999.0*, "min_lon" can be larger or smaller than "max_lon".
  - min_lat [REAL; IN]: Minimum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between -*90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be -*999999.0*. The value of -*999999.0* indicates that the component model wants C-Coupler to calculate "min_lat" automatically. Please note that, "min_lat" calculated by C-Coupler may be incorrect. When "min_lat" has been set to -*90* degrees (or *–PI/2*), it indicates that the global grid covers the South Pole.

- max_lat [REAL; IN]: Maximum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be *-999999.0*. The value of *-999999.0* indicates that the component model wants C-Coupler to calculate "max_lat" automatically. Please note that, "max_lat" calculated by C-Coupler may be incorrect. When "max_lat" has been set to *90* degrees (or *PI/2*), it indicates that the global grid covers the North Pole. When both "min_lat" and "max_lat" are not *-999999.0*, "min_lat" must be smaller than "max_lat".
- center_lon [REAL, DIMENSION(:); IN]: The longitude (X) value of the center of each grid cell. The array size of "center_lon" must be the same with "num_local_cells".
- center_lat [REAL, DIMENSION(:); IN]: The latitude (Y) value of the center of each grid cell. The array size of "center_lat" must be the same with "num_local_cells".
- mask [INTEGER, DIMENSION(:), OPTIONAL; IN]: The mark to specify each local grid cell active or not. The "mask" value of 1 means active while value of 0 means inactive. The array size of "mask" must be the same with "num_local_cells". When "mask" is not provided, it means all local grid cells are active.
- area [REAL, DIMENSION(:), OPTIONAL; IN]: The area of each local grid cell. The array size of "area" must be the same with "num_local_cells". The unit of "area" should be consistent with "center_lon".
- vertex_lon [REAL, DIMENSION(:,:), OPTIONAL; IN]: The longitude (X) values of the vertexes of each local grid cell. Corresponding to "center_lon", "vertex_lon" must have an additional lowest dimension whose size is the maximum number of vertexes. "-999999.0" can be used to specify the missing values.
- vertex_lat [REAL, DIMENSION(:,:), OPTIONAL; IN]: The latitude (Y) values of the vertexes of each grid cell. Corresponding to "center_lat", "vertex_lat" must have an additional lowest dimension whose size is the maximum number of vertexes. "-999999.0" can be used to specify the missing values. "vertex_lon" and "vertex_lat" must be provide at the same time and have the same size of the lowest dimension.
- decomp_name [CHARACTER, OPTIONAL; IN]: The name of the new parallel decomposition to be registered at the same time. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
- decomp_id [CHARACTER, OPTIONAL; OUT]: used to return the ID of the new parallel decomposition to be registered at the same time. "decomp_id" and "decomp_name" must be provided or not provided at the same time.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

- **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to "comp_id" using the local grid data provided by each process of the given component model and returns the ID of the horizontal grid when the registration succeeds. It is similar to the API CCPL_register_H2D_grid_via_global_data, but using the local grid data of the current MPI process for grid registration. The local grid data are described based on the parameters "num_local_cells" and "local_cells_global_index". This API can also register the corresponding parallel decomposition on the new horizontal grid at the same time.

All MPI processes of the component model are required to call this API at the same time, with consistent parameters (the global information including “grid_name”, “edge_type”, “coord_unit”, “cyclic_or_acyclic”, “grid_size”, “min_lon”, “max_lon”, “min_lat” and “max_lat” must be the same across all MPI processes). This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.3 CCPL_register_H2D_grid_via_file

- **Integer FUNCTION CCPL_register_H2D_grid_via_file(comp_id, grid_name, data_file_name, annotation)**
  - return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.
  - data_file_name [CHARACTER; IN]: The name of the file (only NetCDF file currently) with grid data. The corresponding file should be included under the directory grids_weights (Figure 23, Section 3), and “data_file_name” should not include directory. “data_file_name” has a maximum length of 1000 characters. Please refer to the examples section for the detailed requirements for the grid data file.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to “comp_id” based on the grid data in a file and returns the ID of the horizontal grid when the registration succeeds. This API first reads the grid data available in the file and next calls the function CCPL_register_H2D_grid_via_global_data to further register the new horizontal grid. Therefore, the grid data in the file must satisfy the corresponding rules of CCPL_register_H2D_grid_via_global_data except that there is no restriction of dimensions of the variables in the grid data file.

All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

C-Coupler employs a NetCDF format of grid data files similar to the SCRIP tool that is mainly for generating remapping weights. Here we give 3 examples about the NetCDF format.

1) Example 1

Example1 is about a longitude-latitude grid. Dimension “lon” in the grid data file corresponds to the parameter “dim1_size” of CCPL_register_H2D_grid_via_global_data; dimension “lon” corresponds to the parameter “dim2_size”; file variable “grid_center_lon” corresponds to the parameter “center_lon”; file variable “grid_center_lat” corresponds to the parameter “center_lat”; file variable “grid_corner_lon” corresponds to the parameter “vertex_lon”; variable “grid_corner_lat” corresponds to the parameter “vertex_lat”; file variable “grid_imask” corresponds to the parameter “mask”; the unit of “grid_center_lon”/“grid_center_lat”/

“grid_corner_lon”/“grid_corner_lat” corresponds to the parameter “unit”; each of remaining variables and global attributes corresponding to the parameter of CCPL_register_H2D_grid_via_global_data with the same name.

2) Example 2

Example2 is similar to example1, with the difference that exmaple2 enumerates the coordinate values of each grid cell.

3) Example 3

```
dimensions:
        lon = 128 ;
        lat = 60 ;
        num_vertexes_lon = 2 ;
        num_vertexes_lat = 2 ;
variables:
        float grid_center_lon(lon) ;
                grid_center_lon:long_name = "longitude" ;
                grid_center_lon:unit = "degrees" ;
        float grid_corner_lon(lon, num_vertexes_lon) ;
                grid_corner_lon:long_name = "longitude" ;
                grid_corner_lon:unit = "degrees" ;
        float grid_center_lat(lat) ;
                grid_center_lat:long_name = "latitude" ;
                grid_center_lat:unit = "degrees" ;
        float grid_corner_lat(lat, num_vertexes_lat) ;
                grid_corner_lat:long_name = "latitude" ;
                grid_corner_lat:unit = "degrees" ;
        int grid_imask(lat, lon) ;
        float area(lat, lon) ;
// global attributes:
                :edge_type = "LON_LAT" ;
                :cyclic_or_acyclic = "cyclic" ;
                :min_lon = 0. ;
                :max_lon = 360. ;
                :min_lat = -90. ;
                :max_lat = 90. ;
```

Example1

```
dimensions:
        lon = 128 ;
        lat = 60 ;
        num_vertexes_lon = 4 ;
        num_vertexes_lat = 4 ;
variables:
        float grid_center_lon(lat, lon) ;
                grid_center_lon:long_name = "longitude" ;
                grid_center_lon:unit = "degrees" ;
        float grid_corner_lon(lat, lon, num_vertexes_lon) ;
                grid_corner_lon:long_name = "longitude" ;
                grid_corner_lon:unit = "degrees" ;
        float grid_center_lat(lat, lon) ;
                grid_center_lat:long_name = "latitude" ;
                grid_center_lat:unit = "degrees" ;
        float grid_corner_lat(lat, lon, num_vertexes_lat) ;
                grid_corner_lat:long_name = "latitude" ;
                grid_corner_lat:unit = "degrees" ;
        int grid_imask(lat, lon) ;
        float area(lat, lon) ;

// global attributes:
                :edge_type = "LON_LAT" ;
                :cyclic_or_acyclic = "cyclic" ;
                :min_lon = 0. ;
                :max_lon = 360. ;
                :min_lat = -90. ;
                :max_lat = 90. ;
```

Example2

```
dimensions:
        grid_size = 7680 ;
        num_vertexes_lon = 4 ;
        num_vertexes_lat = 4 ;
variables:
        float grid_center_lon(grid_size) ;
                grid_center_lon:long_name = "longitude" ;
                grid_center_lon:unit = "degrees" ;
        float grid_corner_lon(grid_size, num_vertexes_lon) ;
                grid_corner_lon:long_name = "longitude" ;
                grid_corner_lon:unit = "degrees" ;
        float grid_center_lat(grid_size) ;
                grid_center_lat:long_name = "latitude" ;
                grid_center_lat:unit = "degrees" ;
        float grid_corner_lat(grid_size, num_vertexes_lat) ;
                grid_corner_lat:long_name = "latitude" ;
                grid_corner_lat:unit = "degrees" ;
        int grid_imask(grid_size) ;
        float area(grid_size) ;

// global attributes:
                :edge_type = "LON_LAT" ;
                :cyclic_or_acyclic = "cyclic" ;
                :min_lon = 0. ;
                :max_lon = 360. ;
                :min_lat = -90. ;
                :max_lat = 90. ;
```

Example3

Example3 is similar to example1, with the difference that the dimensions “lon” and “lat” are replaced by “grid_size” in exmaple3. Dimension “grid_size” in the grid data file corresponds to parameter “dim1_size” of CCPL_register_H2D_grid_via_global_data, while parameter “dim2_size” should be 0.

2.4.4 CCPL_register_H2D_grid_from_another_component

- **Integer FUNCTION CCPL_register_H2D_grid_from_another_component(comp_id, grid_name, annotation)**
  - return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

- **Examples**

The specification of an H2D grid in another component model corresponding to one entry in section “*local_grids*” of the *coupling connection configuration file*. “local_grid_name” means the name of the new H2D grid to be registered. “another_comp_full_name” means the full name of another component model and “another_comp_grid_name” means the name of the corresponding H2D grid in another component model.

```
<local_grids>
    <entry          local_grid_name="remote_atm_grid"          another_comp_full_name="gamil"
another_comp_grid_name="gamil_H2D_grid" />
</local_grids>
```

2.4.5 CCPL_register_H2D_grid_without_data

- **Integer FUNCTION CCPL_register_H2D_grid_without_data(comp_id, grid_name, grid_size, annotation)**
  - return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.
  - grid_size [INTEGER; IN]: The size of the horizontal grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the

corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a horizontal grid without grid data. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.6 CCPL_register_V1D_Z_grid_via_model_data

- **Integer FUNCTION CCPL_register_V1D_Z_grid_via_model_data(comp_id, grid_name, coord_unit, coord_values, annotation)**
  - return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - coord_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
  - coord_values [REAL, DIMENSION(:); IN]: The coordinate values of the vertical Z grid. The size of the vertical grid is determined by the array size of "coord_values". The values in "coord_values" must be in an ascending or descending order.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a vertical (V1D) Z grid of the component model corresponding to "comp_id" using the global grid data provided by the model and returns the ID of the vertical Z grid when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.7 CCPL_register_V1D_SIGMA_grid_via_model_data

- **Integer FUNCTION CCPL_register_V1D_SIGMA_grid_via_model_data(comp_id, grid_name, coord_unit, P0, sigma_values, annotation)**
  - return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - coord_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
  - P0 [REAL; IN]: The pressure value corresponding to the top level of the SIGMA grid
  - sigma_values [REAL, DIMENSION(:); IN]: The SIGMA values of the vertical SIGMA

grid. The size of the vertical grid is determined by the array size of “sigma_values”. The values in “sigma_values” must be in an ascending or descending order.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

- **Description of this API**

This API registers a vertical (V1D) SIGMA grid of the component model corresponding to “comp_id” using the global grid data provided by the model and returns the ID of the vertical SIGMA grid when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended. The pressure (*p*) vertical coordinate values of a 3-D grid are calculated as *p(i,j,k)=sigma_values(k)*PS(i,j)+(1-sigma_values(k))*P0*, where *PS* means the current surface pressure.

2.4.8 CCPL_register_V1D_HYBRID_grid_via_model_data

- **Integer FUNCTION CCPL_register_V1D_HYBRID_grid_via_model_data(comp_id, grid_name, coord_unit, P0, coef_A, coef_B, annotation)**
  - return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.
  - coord_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
  - P0 [REAL; IN]: The pressure value corresponding to the top level of the HYBRID grid.
  - coef_A [REAL, DIMENSION(:); IN]: The values of coefficients A of the vertical HYBRID grid. The size of the vertical grid is determined by the array size of “coef_A”.
  - coef_B [REAL, DIMENSION(:); IN]: The values of coefficients B of the vertical HYBRID grid. “coef_B” must have the same array size with “coef_A”. The values in “coef_B” must be in an ascending or descending order. “coef_B” works similarly with the sigma values in a vertical SIGMA grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

- **Description of this API**

This API registers a vertical (V1D) HYBRID grid of the component model corresponding to “comp_id” using the global grid data provided by the model and returns the ID of the vertical HYBRID grid when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended. The pressure (*p*) vertical coordinate values of a 3-D grid are calculated as *p(i,j,k)=coef_A(k)*P0+coef_B(k)*PS(i,j)*, where *PS* means the current surface pressure. This formula is consistent with the atmosphere model CAM (please refer to http://www.cesm.ucar.edu/models/atm-cam/docs/usersguide/node25.html).

2.4.9 CCPL_register_V1D_grid_without_data

- **Integer FUNCTION CCPL_register_V1D_grid_without_data(comp_id, grid_name, coord_unit, grid_size, annotation)**
  - return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - coord_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
  - grid_size [INTEGER; IN]: The size of the grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API registers a vertical (V1D) grid of the component model corresponding to "comp_id" without grid data and returns the ID of the vertical grid when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.10 CCPL_register_1D_normal_grid_without_data

- **Integer FUNCTION CCPL_register_1D_normal_grid_without_data(comp_id, grid_name, grid_size, annotation)**
  - return value [INTEGER; OUT]: The ID of the new 1-D grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - grid_size [INTEGER; IN]: The size of the grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API registers a 1-D grid of the component model corresponding to "comp_id" without grid data and returns the ID of the 1-D grid when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters.

This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.11 CCPL_register_MD_grid_via_multi_grids

- **Integer FUNCTION CCPL_register_MD_grid_via_multi_grids(comp_id, grid_name, sub_grid1_id, sub_grid2_id, sub_grid3_id, mask, annotation)**
  - return value [INTEGER; OUT]: The ID of the new multi-dimension (MD) grid.
  - comp_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
  - grid_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - sub_grid1_id [INTEGER; IN]: The ID of the first sub grid that has already been registered. The first sub grid must be associated with the given component model.
  - sub_grid2_id [INTEGER; IN]: The ID of the second sub grid that has already been registered. The second sub grid must be also associated with the given component model, and do not have common coordinates with the first sub grid.
  - sub_grid3_id [INTEGER, OPTIONAL; IN]: The ID of the third sub grid that has already been registered. The third sub grid must be also associated with the given component model, and do not have common coordinates with the first and second sub grids.
  - mask [INTEGER, DIMENSION(:), OPTIONAL; IN]: The mark to specify each grid cell active or not. The "mask" value of 1 means active while value of 0 means inactive. The array size of "mask" must be the size of the multiple-dimension grid. When "mask" is not provided, it means all grid cells are active.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API registers a new multi-dimension (MD) grid using several grids that have already been registered and returns the ID of the new grid when the registration succeeds. The new grid corresponds to the same component model with all sub grids. All MPI processes of the given component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.12 CCPL_register_mid_point_grid

- **SUBROUTINE CCPL_register_mid_point_grid (level_3D_grid_id, mid_3D_grid_id, mid_V1D_grid_id, mask, annotation)**
  - level_3D_grid_id [INTEGER; IN]: The ID of the interface-level grid that has already been registered to C-Coupler. The interface-level grid is a 3-D grid consisting of a horizontal grid and a vertical grid. The interface-level grid cannot be the middle-point grid of another grid.
  - mid_3D_grid_id [INTEGER; OUT]: Return the ID of the middle-point grid.
  - mid_V1D_grid_id [INTEGER; OUT]: Return the ID of the vertical sub grid of the

  middle-point grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers the middle-point grid of an interface-level grid (3-D) and returns the ID of the middle-point grid and the ID of the vertical sub grid of the middle-point grid. These two newly registered grids are associated to the same component model with the interface-level grid. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.13 CCPL_set_3D_grid_variable_surface_field

- **SUBROUTINE CCPL_set_3D_grid_variable_surface_field(grid_id, field_id, annotation)**
  - grid_id [INTEGER; IN]: The ID of the 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a vertical grid that is a SIGMA grid or HYBRID grid.
  - field_id [INTEGER; IN]: The ID of the surface field instance that has already been registered to C-Coupler. This field instance must be on the horizontal sub grid of the 3-D grid, and corresponds to the same component model with the 3-D grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers the dynamic surface field for a 3-D grid. Dynamic surface field means that its values change with the time integration, like the surface pressure in an atmospheric model. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.14 CCPL_set_3D_grid_constant_surface_field

- **SUBROUTINE CCPL_set_3D_grid_constant_surface_field(grid_id, field_id, annotation)**
  - grid_id [INTEGER; IN]: The ID of the 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a vertical grid that is a SIGMA grid or HYBRID grid.
  - field_id [INTEGER; IN]: The ID of the surface field instance that has already been registered to C-Coupler. This field instance must be on the horizontal sub grid of the 3-D grid, and corresponds to the same component model with the 3-D grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API registers the static surface field for a 3-D grid. Static surface field means that its values keeps constant with the time integration, like the depth of the bottom of oceans in an ocean model. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.15 CCPL_set_3D_grid_external_surface_field

- **SUBROUTINE CCPL_set_3D_grid_external_surface_field(grid_id, annotation)**
  - grid_id [INTEGER; IN]: The ID of the 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a vertical grid that is a SIGMA grid or HYBRID grid.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API declares that the surface field of a 3-D grid is external. External surface field means that its values are determined by the surface field of another 3-D grid in model coupling. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.16 CCPL_set_3D_grid_3D_vertical_coord_field

- **SUBROUTINE CCPL_set_3D_grid_3D_vertical_coord_field(grid_id, field_id, label, annotation)**
  - grid_id [INTEGER, IN]: The ID of a 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a normal vertical grid without coordinate values (registered by the API CCPL_register_1D_normal_grid_without_data).
  - field_id [INTEGER, IN]: The ID of a 3-D field instance that contains the 3-D values of the vertical coordinate of the 3-D grid. 3-D field instance should be on the 3-D grid.
  - label [CHARACTER, IN]: a string for specifying whether the values of the 3-D field are constant (corresponding to the value of “constant”) or variable (corresponding to the value of “variable”) in model integration. When the values are variable, the vertical remapping weights corresponding to the 3-D grid will be dynamically updated in model integration.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API sets the 3-D values of the vertical coordinate of the 3-D grid based on a 3-D field instance. The 3-D values can be either constant or variable in model integration. When the 3-D values are variable, the vertical remapping weights corresponding to the 3-D grid will be dynamically updated in model integration. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.4.17 CCPL_get_grid_size

- **integer FUNCTION CCPL_get_grid_size(grid_id, annotation)**
  - return value [INTEGER; OUT]: The size of the grid.
  - grid_id [INTEGER; IN]: The ID of the grid that has already been registered to C-Coupler.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API returns the size of a grid that has already been registered to C-Coupler.

2.4.18 CCPL_get_grid_id

- **integer FUNCTION CCPL_get_grid_id(comp_id, grid_name, annotation)**
  - return value [INTEGER; OUT]: The ID of the new grid.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - grid_name [CHARACTER; IN]: The name of the grid. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API will return the ID of the corresponding grid when it has already been registered to C-Coupler or return -1 when the corresponding grid has not been registered.

2.4.19 CCPL_get_H2D_grid_data

- **SUBROUTINE CCPL_get_H2D_grid_data (grid_id, decomp_id, label, grid_data, annotation)**
  - grid_id [INTEGER; IN]: The ID of the given horizontal (H2D) grid that has already been registered to C-Coupler.
  - decomp_id [INTEGER; IN]: The ID of a given parallel decomposition on the horizontal grid. When local grid data is wanted, a valid "decomp_id" should be given, and the corresponding parallel decomposition must be associated with the given horizontal grid; when global grid data is wanted, "decomp_id" should be set to -1.
  - label [CHARACTE; IN]: The label of the grid data. The valid labels currently supported include "lon", "lat", and "mask". "lon" means the longitude (X) values of all grid cells. "lat" means the latitude (Y) values of all grid cells. "mask" means the mask values of all grid cells.
  - grid_data [REAL|INTEGER, DIMENSION(:); out]: Return the grid data. The array size of "grid_data" must be consistent with given parallel decomposition. The data type of "grid_data" must be consistent with "label" (for example, when "label" is "mask", "grid_data" must be an integer array).

- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API gets the grid data of a horizontal (H2D) grid that has already been registered to C-Coupler.

2.5 APIs for parallel decomposition management

To make a model accelerated by a modern high-performance computer that generally includes a large number of processor cores, the model needs to be parallelized with MPI, where the domains of model grids are decomposed into separate subdomains for parallel integration. To accommodate the parallel integration of component models and to make model coupling handled in parallel, C-Coupler also manages such kind of decomposition (called parallel decomposition) and provides APIs to enable active component models to register their parallel decompositions to C-Coupler.

Currently, C-Coupler only supports the parallel decompositions on horizontal grids, which means that the parallel decomposition further on vertical grids are not supported yet. Therefore, a parallel decomposition is associated with a horizontal grid, so as to be associated with the component model corresponding to the horizontal grid. The keyword for a parallel decomposition can be expressed as <*ID of component model*, *parallel decomposition name*>. Therefore, different parallel decompositions in the same component model cannot have the same name, while the parallel decompositions in different component models can have the same name. There could be multiple parallel decompositions on the same horizontal grid.

A parallel decomposition on a horizontal grid is described through enumerating global grid cell indexes of the local grid cells assigned to each process of the corresponding component model. A valid global grid cell index should be between 1 and the size of the horizontal grid. For the local grid cells that are unnecessary to be considered in model coupling (for example, the land only grid cells in an ocean model), the corresponding values of global grid cell index can be set to a C-Coupler pre-defined variable *CCPL_NULL_INT*, to save some overhead in model coupling.

Table 5 APIs for parallel decomposition management.

No.	API	Brief description
1	CCPL_register_normal_parallel_decomp	Register a parallel decomposition on a horizontal grid
2	CCPL_register_chunk_parallel_decomp	Register chunked parallel decomposition
3	CCPL_get_decomp_num_chunks	Get number of chunks
4	CCPL_get_decomp_chunks_size	Get size of each chunk

The local grid cells assigned to a process can be divided into a number of subsets, each of which is called a chunk. A parallel decomposition with chunks is called chunked parallel decomposition. When registering a field instance on a chunked parallel decomposition, different chunks of the field instance can correspond to different memory spaces in the same process.

Table 5 lists out the API for parallel decomposition management as well as its brief descriptions.

2.5.1 CCPL_register_normal_parallel_decomp

- **integer FUNCTION CCPL_register_normal_parallel_decomp(decomp_name, grid_id, num_local_cells, local_cells_global_index, annotation)**
  - return value [INTEGER; OUT]: The ID of the parallel decomposition.
  - decomp_name [CHARACTER; IN]: The name of the parallel decomposition. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - grid_id [INTEGER; IN]: The ID of the corresponding horizontal (H2D) grid that has already been registered to C-Coupler.
  - num_local_cells [INTEGER; IN]: The number of local grid cells (≥0) in the parallel decomposition of the current MPI process.
  - local_cells_global_index [INTEGER, DIMENSION(:); IN]: The global index of the local grid cells in the parallel decomposition of the current MPI process. The array size of "local_cells_global_index" cannot be smaller than "num_local_cells". Each value in "local_cells_global_index" must be *CCPL_NULL_INT* or a value between 1 and the grid size.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a new parallel decomposition of a horizontal grid ("grid_id") among all MPI processes of the component model corresponding to "grid_id", and returns the ID of the new parallel decomposition when the registration succeeds. The new parallel decomposition corresponds to the same component model with "grid_id". All MPI processes of the component model are required to call this API at the same time, with the same "decomp_name" and the same horizontal grid. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.5.2 CCPL_register_chunk_parallel_decomp

- **integer FUNCTION CCPL_register_chunk_parallel_decomp (decomp_name, grid_id, num_chunks, chunks_size, local_cells_global_index, annotation)**
  - return value [INTEGER; OUT]: The ID of the parallel decomposition with chunks.
  - decomp_name [CHARACTER; IN]: The name of the parallel decomposition. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - grid_id [INTEGER; IN]: The ID of the corresponding horizontal (H2D) grid that has already been registered to C-Coupler.
  - num_chunks [INTEGER; IN]: The number of chunks (≥0) of the parallel decomposition in the current MPI process.
  - chunk_size [INTEGER, DIMENSION(:); IN]: The size of each chunk in the parallel decomposition of the current MPI process, i.e., the number of local grid cells (>0) in each chunk.
  - local_cells_global_index [INTEGER, DIMENSION(:); IN]: The global index of each

local grid cell of the parallel decomposition in the current MPI process. The array size of “local_cells_global_index” cannot be smaller than the total size of all chunks in the current process.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a new parallel decomposition with chunks corresponding to the horizontal grid specified via “grid_id”, and returns the ID of the new parallel decomposition when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with the same “decomp_name” and the same horizontal grid. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.5.3 CCPL_get_decomp_num_chunks

- **integer FUNCTION CCPL_get_decomp_num_chunks (decomp_id, annotation)**
  - return value [INTEGER; OUT]: The number of chunks in the specified parallel decomposition in the current process.
  - decomp_id [INTEGER; IN]: The ID of a parallel decomposition.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the number of chunks in the specified parallel decomposition in the current process.

2.5.4 CCPL_get_decomp_chunks_size

- **SUBROUTINE CCPL_get_decomp_chunks_size (decomp_id, chunks_size, annotation)**
  - decomp_id [INTEGER; IN]: The ID of a given parallel decomposition.
  - chunk_size [INTEGER, DIMENSION(:); OUT]: It returns the sizes of all chunks in the parallel decomposition in the current MPI process.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the sizes of all chunks in the parallel decomposition in the current MPI process.

2.6 APIs for coupling field management

C-Coupler employs a concept of coupling field instance to manage coupling fields. A coupling field instance includes a set of META information and a memory buffer that keeps the data values of an instance of a coupling field. A field instance is associated with a unique component model, a unique grid and a unique parallel decomposition. Moreover, an attribute of "buf_mark" is employed in each field instance, to separate the multiple field instances that are in the same component model, on the same grid and on the same parallel decomposition (for example, as land surface, oceans and sea ice are under the bottom of atmosphere, an atmosphere model may receive multiple coupling field instances of surface temperature each of which is from a land surface model, an ocean model and a sea ice model respectively). Therefore, the keyword for a field instance can be expressed as <ID of component model, ID of grid, ID of parallel decomposition, buf_mark>. For a scalar field instance that is not on a grid, the corresponding grid ID and parallel decomposition ID should be set to *-1*. Based on C-Coupler APIs, a component model can register field instances to C-Coupler, in order for providing, obtaining and remapping coupling fields in model coupling. An internal model field instance that will not be used in model coupling can also be registered to C-Coupler for exact restart capability.

A field instance can be on a chunked parallel decomposition. There are APIs for registering field instances on a chunked parallel decomposition.

Table 6 lists out the APIs for coupling field management as well as their brief descriptions.

Table 6 APIs for coupling field management.

No.	API	Brief description
1	CCPL_register_field_instance	Register a field instance
2	CCPL_start_chunk_field_instance_registration	Start registering a chunk field instance
3	CCPL_register_chunk_of_field_instance	Register a chunk field instance
4	CCPL_finish_chunk_field_instance_registration	Finish registering a chunk field instance

2.6.1 CCPL_register_field_instance

- **integer FUNCTION CCPL_register_field_instance(data_buf, field_name, decomp_id, comp_or_grid_id, buf_mark, usage_tag, field_unit, annotation)**
  - Return value [INTEGER; OUT]: The ID of the new field instance.
  - data_buf [REAL or INTEGER, no DIMENSION or DIMENSION|(:), (:,:), (:,:,:) or (:,:,:,:)|; INOUT]: The model data buffer corresponding to the field instance.
  - field_name [CHARACTER; IN]: The name of the coupling field. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'. Different field instances can share the same "field_name".
  - decomp_id [INTEGER; IN]: If the field instance is a scalar variable or the field instance is not on a grid with horizontal sub grid (for example, the field instance is only on a vertical grid), "decomp_id" is *-1*; otherwise, "decomp_id" is the ID of the corresponding parallel decomposition that has already been registered to C-Coupler.

- comp_or_grid_id [INTEGER; IN]: If the field instance is a scalar variable, “comp_or_grid_id” is the ID of the corresponding component model; otherwise, “comp_or_grid_id” is the ID of the corresponding grid that has already been registered to C-Coupler. When “comp_or_grid_id” is the ID of a grid, “decomp_id” and “comp_or_grid_id” must correspond to the same component model, and the horizontal grid corresponding to “decomp_id” must be a sub grid of the grid corresponding to “comp_or_grid_id”.
- buf_mark [INTEGER; IN]: a mark used to separate different field instances with the same field name, the same “decomp_id” and the same “comp_or_grid_id”. “buf_mark” must be a non-negative integer.
- usage_tag [Integer, OPTIONAL; IN]: used to specify how to use the given field instance, i.e., used in model coupling, or used in restart write/read. Currently, there are three options for the value of “*usage_tag*”: CCPL_TAG_CPL, CCPL_TAG_REST, CCPL_TAG_CPL_REST. When “*usage_tag*” is CCPL_TAG_CPL or CCPL_TAG_CPL_REST, the given field instance will be used in model coupling (further used in import/export or remapping interfaces), and the configuration file “public_field_attribute.xml” (Please refer to Section 3.2 for details) must include an entry corresponding to “field_name”. When “*usage_tag*” is CCPL_TAG_REST or CCPL_TAG_CPL_REST, the given field instance will be involved in restart write/read. When “*usage_tag*” is not specified, C-Coupler will use CCPL_TAG_CPL as the default value.
- field_unit [CHARACTER, OPTIONAL; IN]: The unit of the field instance. Default unit specified in the configuration file “public_field_attribute.xml” (Please refer to Section 3.2 for details) will be used when “*field_unit*” is not specified when calling this API.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API uses a model data buffer to register a coupling field instance, and returns the ID of the new coupling field instance when the registration succeeds. “field_name”, “decomp_id”, “comp_or_grid_id” and “buf_mark” are keywords of a coupling field instance, which means two coupling field instances cannot share the same “field_name”, “decomp_id”, “comp_or_grid_id” and “buf_mark”. The parallel decomposition corresponds to the same component model with “comp_or_grid_id”. The array size of “data_buf” must be the same as the required. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.6.2 CCPL_start_chunk_field_instance_registration

• **integer FUNCTION CCPL_start_chunk_field_instance_registration (field_name, decomp_id, grid_id, buf_mark, usage_tag, field_unit, annotation)**
- return value [INTEGER; OUT]: The ID of the new field instance with chunks.
- field_name [CHARACTER; IN]: The name of the new field instance. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’. Different

field instances can share the same “field_name”. A “field_name” is legal only when there is a corresponding entry in the configuration file “public_field_attribute.xml”.

- decomp_id [INTEGER; IN]: The ID of the corresponding parallel decomposition that has already been registered to C-Coupler3 via API “*CCPL_register_chunk_parallel_decomp*”.
- grid_id [INTEGER; IN]: The ID of grid corresponding to the new field instance.
- buf_mark [INTEGER; IN]: A mark used to separate different coupling field instances with the same field name, the same “decomp_id” and the same “grid_id”. “buf_mark” must be a non-negative integer.
- usage_tag [INTEGER; OPTIONAL; IN]：A mark used to specify how to use the new registry chunk field instance, either for model coupling or for restart writes/reads. Currently, the value of "usage_tag" has three options: CCPL_TAG_CPL, CCPL_TAG_REST, and CCPL_TAG_CPL_REST. When "usage_tag" is CCPL_TAG_CPL or CCPL_TAG_CPL_ REST, the new registered field instance will be used for model coupling, and the configuration file "public_field_attribute.xml" must contain the entry corresponding to "field_name". When "usage_tag" is CCPL_TAG_REST or CCPL_TAG_CPL_REST, the new registered variable instance participates in restart writes/reads.
- field_unit [CHARACTER, OPTIONAL; IN]: The unit of the coupling field instance. Default unit specified in the configuration file “public_field_attribute.xml” will be used when “field_unit” is not specified when calling this API.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API starts the registration of a field instance with chunks, and returns the ID of the new new field instance when the registration succeeds. “field_name”, “decomp_id”, “grid_id” and “buf_mark” are keywords of a chunk field instance, which means two coupling field instances cannot share the same “field_name”, “decomp_id”, “grid_id” and “buf_mark”. The parallel decomposition corresponds to the same component model with “grid_id”. All MPI processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.6.3 CCPL_register_chunk_of_field_instance

• **SUBROUTINE CCPL_register_chunk_of_field_instance (data_buf, field_instance_id, annotation)**

- data_buf [REAL or INTEGER, no DIMENSION or DIMENSION|(:), (:,:), (:,:,:) or (:,:,:,:)|; INOUT]: The model data buffer of one chunk of a field instance with chunks.
- field_instance_id [INTEGER; IN]: The ID of the field instance.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API sets the data buffer of one chunk of the corresponding field instance. The array size of “data_buf” must be the same as the required array size. When a field instance has multiple chunks in a MPI process, this process needs to call this API multiple times each of which specifies the data buffer of a chunk. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.6.4 CCPL_finish_chunk_field_instance_registration

- **SUBROUTINE CCPL_finish_chunk_field_instance_registration (field_instance_id, annotation))**
  - field_instance_id [INTEGER; IN]: The ID of the field instance with chunks.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API finishes the registration of a field instance with chunks. To register a field instance with chunks, one call of “*CCPL_start_chunk_field_instance_registration*”, multiple call of “*CCPL_register_chunk_of_field_instance*” and one call of “*CCPL_finish_chunk_field_instance_registration*” should be performed successively. All MPI processes of the component model are required to call “*CCPL_finish_chunk_field_instance_registration*” at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.7 APIs for coupling interface management

An active component model can handle coupling field instances through coupling interfaces. The keyword of a coupling interface can be expressed as <*ID of the component model*, *interface name*>. Therefore, different coupling interfaces in the same component model cannot have the same interface name, while the coupling interfaces in different component models can have the same interface name.

Coupling interfaces are classified into three categories: import interfaces, export interfaces and remap interfaces. An import interface enables a component model to obtain coupling field instances from itself or other component models. Specifically, an import interface can be specified to obtain instantaneous or averaged coupling field instances. An export interface enables a component model to provide a number of coupling field instances to the coupled model. A remap interface enables a component model to remap its coupling fields from a source grid to a target grid. There are two detailed kinds of remap interfaces: normal remap interface and fraction based remap interface. A normal remap interface directly interpolates coupling field instances from the source grid to the target grid, while a fraction based remap interface will first adjust the values of coupling field instances on the source grid based on the source fraction before remapping and finally adjust the values of coupling field instances on the target grid based on the target fraction after remapping (the source fraction is also remapped from the source grid to the target grid, in order to produce the target fraction at the same time). Fraction based remap interfaces are generally necessary for guaranteeing conservation in model coupling.

There are three steps to utilize a coupling interface. The first step is to register the coupling interface, where a timer is required to be specified for controlling the time to execute the coupling interface. The second step is to generate coupling procedures for the coupling interface. The third step is to execute the coupling interface. Although the API to execute a coupling interface can be called at each time step, a coupling interface will be truly executed only when its timer is bypassed or its timer is on. C-Coupler enables to bypass the timer when executing a coupling interface, in order to achieve flexible coupling at the initialization stage of the coupled model. Please note that the timer of a coupling interface cannot be bypassed again if this coupling interface has already been executed with the timer on before, and please note that when the timer of a coupling interface is not bypassed, the coupling interface will be truly executed at most once at each time step, which means that additional API calls for executing the coupling interface at a time step will be neglected.

Table 7 APIs for coupling interface management.

No.	API	Brief description
1	CCPL_register_export_interface	Register a coupling interface for exporting field instances
2	CCPL_register_import_interface	Register a coupling interface for importing field instances
3	CCPL_register_normal_remap_interface	Registering a coupling interface for remapping field instances normally
4	CCPL_register_frac_based_remap_interface	Registering a coupling interface for remapping field instances where faction will be used
5	CCPL_execute_interface_using_id	Execute a coupling interface specified by an interface ID
6	CCPL_execute_interface_using_name	Execute a coupling interface in a component model specified by an interface name
7	CCPL_get_H2D_grid_area_in_remapping_wgts	Get the area of grid cells of a horizontal grid that are calculated in remapping weights calculation in coupling procedure generation for the given coupling field instance of the given coupling interface
8	CCPL_check_is_import_field_connected	Check whether the given import field instance in the given import interface has already been connected through coupling procedure generation
9	CCPL_get_import_fields_sender_time	Get the latest sender model time of all field instances received by an import interface

For a remap interface that does not refer to coupling between different coupling interfaces or different component models, its coupling procedures are generated implicitly by the coupling generator when registering it. Coupling procedures of an export/import interface are also generated by the coupling generator automatically, but will not be generated when registering the interface, because an export/import interface refers to coupling between different coupling interfaces in the same or different component models. To generate coupling procedures of export/import interfaces, the coupling generator will analyze possible connections from export interfaces to import interfaces based on the field name of each coupling field instance. A coupling connection from an export interface to an import interface can be generated only when these two coupling interfaces have common field names. Regarding a field name, C-Coupler allows an export interface to be connected to any number of import interfaces, while forces an import interface be connected from a unique export interface. In other words, each coupling field instance in an import interface must have only one provider. If there are multiple providers for a coupling field instance in an import interface, users must specify how to pick out only one provider through the corresponding configuration file (please refer to Section 3.1). Different coupling field instances in an import interface can have different providers. Coupling procedures of import/export interfaces are generated through explicitly calling the APIs for coupling procedure generation.

An export interface or a remap interface always can be executed successfully, which means no error will be reported when executing an export interface or a remap interface. It may fail to execute an import interface and an error will be reported accordingly, if the coupling procedures of some necessary coupling field instances have not been generated (means that the providers of some necessary coupling field instances have not been found). When registering an import interface (through the API "*CCPL_register_import_interface*"), each import coupling field instance can be specified as necessary or optional. No error will be reported if the providers of some optional coupling field instances have not been found.

Table 7 lists out the APIs for parallel decomposition management as well as their brief descriptions.

2.7.1 CCPL_register_export_interface

- **integer FUNCTION CCPL_register_export_interface(interface_name, num_field_instances, field_instance_IDs, timer_ID, annotation)**
  - Return value [INTEGER; OUT]: The ID of the new export interface.
  - interface_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - num_field_instances [INTEGER; IN]: The number of field instances (>0) exported via this coupling interface.
  - field_instance_IDs [INTEGER, DIMENSION(:); IN]: The ID of the field instances that are exported by this interface. All field instances specified by "field_instance_IDs" must correspond to the same component model. The array size of "field_instance_IDs" cannot be smaller than "num_field_instances". Any two coupling field instances cannot share the same field name.
  - timer_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. Field instances will be exported when the timer is on. "timer_ID" must correspond to the same component model with all field instances.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a new coupling interface that can export a number of field instances produced by the corresponding component model, and returns the ID of the new interface when the registration succeeds. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.7.2 CCPL_register_import_interface

- **integer FUNCTION CCPL_register_import_interface(interface_name, num_field_instances, field_instance_IDs, timer_ID, inst_or_aver, necessity, annotation)**
  - return value [INTEGER; OUT]: The ID of the new import interface.
  - interface_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - num_field_instances [INTEGER; IN]: The number of field instances imported via this coupling interface. "num_field_instances" must be larger than 0.
  - field_instance_IDs [INTEGER, DIMENSION(:); IN]: The ID of the field instances that will be imported by this interface. All field instances specified by "field_instance_IDs" must correspond to the same component model. The array size of "field_instance_IDs" cannot be smaller than "num_field_instances". Any two coupling field instances cannot share the same field name.
  - timer_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. Field instances will be imported when the timer is on. "timer_ID" must correspond to the same component model with all field instances.
  - inst_or_aver [INTEGER; IN]: The mark for specifying using average value or instantaneous value when importing the field instances. The value of *1* means using average value, while the value of *0* means using instantaneous value.
  - necessity [INTEGER, DIMENSION(:), OPTIONAL; IN]: an array each element of which specifies whether the corresponding import field instance is necessary (value of 1) or optional (value of 0). When this parameter is not specified, all import field instances are necessary. When executing an import interface, if a necessary import field instance has not been connected (the provider has not been found and the corresponding coupling procedures have not been generated), the whole model run will be stopped with an error report.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API registers a new coupling interface that enables a corresponding component model to import a number of field instances from itself or other component models, and returns the ID of the new interface when the registration succeeds. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.7.3 CCPL_register_normal_remap_interface

- **integer FUNCTION CCPL_register_normal_remap_interface(interface_name, num_field_instances, field_instance_IDs_source, field_instance_IDs_target, timer_ID, inst_or_aver, annotation)**
  - Return value [INTEGER; OUT]: The ID of the new remapping interface.
  - interface_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - num_field_instances [INTEGER; IN]: The number of field instances remapped via this coupling interface. "num_field_instances" must be larger than 0.
  - field_instance_IDs_source [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the source grids. The array size of "field_instance_IDs_source" cannot be smaller than "num_field_instances". Any two source coupling field instances cannot share the same field name.
  - field_instance_IDs_target [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the target grids. The array size of "field_instance_IDs_target" cannot be smaller than "num_field_instances". "field_instance_IDs_target" must be consistent with "field_instance_IDs_source": the *i*th element in "field_instance_IDs_source" and "field_instance_IDs_target" must correspond to the same field name. All field instances specified by "field_instance_IDs_source" and "field_instance_IDs_target" must correspond to the same component model.
  - timer_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. All source field instances will remapped to target field instances when the timer is on. "timer_ID" must correspond to the same component model with all field instances.
  - inst_or_aver [INTEGER; IN]: The mark for specifying using average value or instantaneous value for remapping. The value of *1* means using average value, while the value of *0* means using instantaneous value.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers a new coupling interface for remapping a number of field instances normally and returns the ID of the new interface when the registration succeeds. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.7.4 CCPL_register_frac_based_remap_interface

- **Integer FUNCTION CCPL_register_frac_based_remap_interface(interface_name, num_field_instances, field_instance_IDs_source, field_instance_IDs_target, timer_ID, inst_or_aver, frac_src, frac_dst, annotation)**
  - return value [INTEGER; OUT]: The ID of the new remapping interface.
  - interface_name [CHARACTER; IN]: The name of the new instance. It has a maximum

length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.

- num_field_instances [INTEGER; IN]: The number of field instances remapped via this coupling interface. “num_field_instances” must be larger than 0.
- field_instance_IDs_source [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the source grids. All source field instances must be on the same horizontal grid and on the same parallel decomposition. The array size of “field_instance_IDs_source” cannot be smaller than “num_field_instances”. Any two source coupling field instances cannot share the same field name.
- field_instance_IDs_target [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the target grids. All target field instances must be on the same horizontal grid and on the same parallel decomposition. The array size of “field_instance_IDs_target” cannot be smaller than “num_field_instances”. “field_instance_IDs_target” must be consistent with “field_instance_IDs_source”: the *i*th element in “field_instance_IDs_source” and “field_instance_IDs_target” must correspond to the same field name. All field instances specified by “field_instance_IDs_source” and “field_instance_IDs_target” must correspond to the same component model.
- timer_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. All source field instances will remapped to target field instances when the timer is on. “timer_ID” must correspond to the same component model with all field instances.
- inst_or_aver [INTEGER; IN]: The mark for specifying using average value or instantaneous value for remapping. The value of 1 means using average value, while the value of 0 means using instantaneous value.
- frac_src [REAL, DIMENSION(:); IN]: The fraction values on the source grid. Its array size must be the same with the field size of each source field instance.
- frac_dst [REAL, DIMENSION(:), OPTIONAL; OUT]: The fraction values on the target grid. Its array size must be the same with the field size of each target field instance.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API registers a new coupling interface that uses the fraction values to remap a number of field instances, and then returns the ID of the new interface when the registration succeeds. The remapping weights for each coupling field must be the same. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

2.7.5 CCPL_execute_interface_using_id

• **logical FUNCTION CCPL_execute_interface_using_id(interface_id, bypass_timer, field_update_status, annotation)**

- Return value [LOGICAL; OUT]: *True* will be returned constantly for C-Coupler. This return value is preserved for future C-Coupler versions.
- interface_id [INTEGER; IN]: The ID of the coupling interface that is to be executed.
- bypass_timer [LOGICAL; IN]: A mark used to specify whether bypassing the timer of the

coupling interface. When "bypass_timer" is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be constantly executed (the field instances will be exported, imported or remapped). When "bypass_timer" is set to *false*, the coupling interface will be executed only when its timer is on.

- field_update_status [INTEGER, DIMENSION(:), OPTIONAL; OUT]: An array where each element is used to record the status that whether the corresponding input field instance (for example, a coupling field instance in an import interface or a target coupling field instance in a remap interface) is updated in the current call of this API. Value of *1* means that the corresponding input field instance is updated while value of *0* means that the corresponding input field instance is not updated.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API executes the given coupling interface, and returns *true* constantly. An export interface or a remap interface always can be executed, while an import interface can be executed only when the coupling procedures for its necessary coupling field instances have been generated. All MPI processes in the corresponding component model are required to call this API at the same time. When "bypass_timer" is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be constantly executed (the field instances will be exported, imported or remapped). Please note that a timer bypassed execution cannot be after a timer non-bypassed execution of the same coupling interface, and the execution of two coupling interfaces that have coupling connections must bypass or not bypass the timers at the same time. When "bypass_timer" is set to *false*, the coupling interface will be executed only when its timer is on, and the given coupling interface will be truly executed at most once at the same time step (in other words, for multiple API calls corresponding to the same coupling interface at the same time step, only the first API call will be truly executed, while the remaining calls will be bypassed).

2.7.6 CCPL_execute_interface_using_name

• **logical FUNCTION CCPL_execute_interface_using_name(component_id, interface_name, bypass_timer, field_update_status, annotation)**

- Return value [LOGICAL; OUT]: If the corresponding coupling interface has been executed successfully, *true* will be returned; otherwise (for example, an import interface fail to obtain the coupling field instances required by the corresponding component model), *false* will be returned.
- component_id [INTEGER; IN]: The ID of the component model corresponding to this interface.
- interface_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
- bypass_timer [LOGICAL; IN]: A mark used to specify whether bypassing the timer of the coupling interface. When "bypass_timer" is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be

constantly executed (the field instances will be exported, imported or remapped). When "bypass_timer" is set to *false*, the coupling interface will be executed only when its timer is on.

- field_update_status [INTEGER, DIMENSION(:), OPTIONAL; OUT]: An array where each element is used to record the status that whether the corresponding input field instance (for example, a coupling field instance in an import interface or a target coupling field instance in a remap interface) is updated in the current call of this API. Value of *1* means that the corresponding input field instance is updated while value of *0* means that the corresponding input field instance is not updated.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API executes the given coupling interface, and returns *true* constantly. An export interface or a remap interface always can be executed, while an import interface can be executed only when the coupling procedures for its necessary coupling field instances have been generated. All MPI processes in the corresponding component model are required to call this API at the same time. When "bypass_timer" is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be constantly executed (the field instances will be exported, imported or remapped). Please note that a timer bypassed execution cannot be after a timer non-bypassed execution of the same coupling interface, and the execution of two coupling interfaces that have coupling connections must bypass or not bypass the timers at the same time. When "bypass_timer" is set to *false*, the coupling interface will be executed only when its timer is on, and the given coupling interface will be truly executed at most once at the same time step (in other words, for multiple API calls corresponding to the same coupling interface at the same time step, only the first API call will be truly executed, while the remaining calls will be bypassed).

2.7.7 CCPL_get_H2D_grid_area_in_remapping_wgts

• **logical FUNCTION CCPL_get_H2D_grid_area_in_remapping_wgts(interface_id, field_index, area_array, annotation)**

- Return value [LOGICAL; OUT]: If the grid area is successfully obtained, *true* will be returned; otherwise, *false* will be returned.
- interface_id [INTEGER; IN]: The ID of a given coupling interface.
- field_index [INTEGER; IN]: The index of a given field instance in the given coupling interface. The index of a field instance in a coupling interface is determined by the order of field instance IDs in the corresponding input arrays in the corresponding API call for registering the coupling interface. The index of the first field instance is *1*.
- area_array [REAL, DIMENSION(:); OUT]: The array for outputting the area values of the corresponding horizontal grid that are calculated in remapping weights calculation. The array size of "area_array" must be no smaller than the required.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API gets the area of grid cells of a horizontal grid that are calculated in remapping weights calculation in coupling procedure generation for the given coupling field instance of the given coupling interface. It may be failed (*false* will be returned) to get the area when the given field instance has not been involved in coupling procedure generation or data interpolation is unnecessary in coupling procedure generation (remapping weights will not be calculated when data interpolation is unnecessary).

2.7.8 CCPL_check_is_import_field_connected

- **logical FUNCTION CCPL_check_is_import_field_connected(import_interface_id, field_instance_id, annotation)**
  - Return value [LOGICAL; OUT]: If the given import field instance in the given import interface has been connected (the provider of the field instance has been found and the corresponding coupling procedures have been generated through coupling procedure generations), *true* will be returned; otherwise, *false* will be returned.
  - import_interface_id [INTEGER; IN]: The ID of the given import interface.
  - field_instance_id [INTEGER; IN]: The ID of the given import field instance in the given import interface.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

  This API checks whether the given import field instance in the given import interface has already been connected through coupling procedure generations.

2.7.9 CCPL_get_import_fields_sender_time

- **SUBROUTINE CCPL_get_import_fields_sender_time(import_interface_id, sender_date, sender_second, sender_elapsed_days, annotation)**
  - import_interface_id [INTEGER; IN]: The ID of the given import interface.
  - sender_date [INTEGER, DIMENSION(:); OUT]: The array for returning the date of the latest sender model time of all field instances received by the given import interface. The array size of "sender_date" cannot be smaller than the number of the field instances of the given import interface. If the given import interface did not receive a field instance before, the given element in "sender_date" will be set to *CCPL_NULL_INT* (0x7FFFFFFF).
  - sender_second [INTEGER, DIMENSION(:); OUT]: The array for returning the second of the latest sender model time of all field instances received by the given import interface. The array size of "sender_second" cannot be smaller than the number of the field instances of the given import interface. If the given import interface did not receive a field instance before, the given element in "sender_second" will be set to *CCPL_NULL_INT* (0x7FFFFFFF).
  - sender_elapsed_days [INTEGER, DIMENSION(:), OPTIONAL; OUT]: The array for returning the number of elapsed day of the latest sender model time (since 0000-01-01) of

all field instances received by the given import interface. The array size of "sender_elapased_days" cannot be smaller than the number of the field instances of the given import interface. If the given import interface did not receive a field instance before, the given element in "sender_elapased_days" will be set to *CCPL_NULL_INT* (0x7FFFFFFF).

- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API gets the latest sender model time of all field instances received by the given import interface.

2.8 APIs for coupling procedure generation

Table 8 lists out the APIs for coupling procedure generation as well as their brief descriptions, which enable to perform a coupling procedure generation for any subset of component models. The coupling procedure generation related to a component model is classified into two modes: individual or family. For the individual mode, only the given component model itself will be considered in coupling procedure generation, while for the family mode, the given component model and its descendant component models can be considered in the same coupling procedure generation. When registering a component model through the API "*CCPL_register_component*", it can be specified to enable or disable the given component model in the family coupling procedure generation of its parent or any ancestor. The API "*CCPL_do_external_coupling_generation*" can do coupling procedure generation regarding to any subset of component models, where either individual or family coupling procedure generation can be specified for each given component model. Moreover, the API "*CCPL_get_configurable_comps_full_names*" enables to flexibly specify a subset of component models in XML configuration file, to cooperate with the API "*CCPL_do_external_coupling_generation*" to further improve the flexibility of coupling procedure generation. Besides partial coupling procedure generations, a global coupling procedure generation will still be performed when root component models are calling the API "*CCPL_end_coupling_configuration*", while a root component model that has been disabled in the family coupling procedure generation will not be involved in the global coupling procedure generation.

As a coupling procedure generation can be performed for any subset of component models, a component model can participate in multiple coupling procedure generations. In other words, the coupling procedures of a component model or even an import/export interface can be incrementally generated through multiple coupling procedure generations. Regarding an import interface in a coupling procedure generation, only the import fields whose coupling procedures have not been generated will be considered in the coupling procedure generation, while the import fields whose coupling procedures have already been generated will be neglected.

Note that, C-Coupler does not consider the coupling connections within the same component model in default. When the user wants coupling within the same component model, the corresponding coupling connections must be specified explicitly through XML configuration files.

Table 8 APIs for coupling procedure generation.

No.	API	Brief description
1	CCPL_do_individual_coupling_generation	Do coupling procedure generation intra the given component model
2	CCPL_do_family_coupling_generation	Do family coupling procedure generation among the given component model and its descendant component models that are not disabled in family coupling procedure generation
3	CCPL_do_external_coupling_generation	Do coupling procedure generation for a given set of component models. Either individual or family coupling procedure generation can be specified for each given component model.
4	CCPL_get_configurable_comps_full_names	Get the full names (as well as the specification of either individual or family coupling procedure generation) of a set of component models from the corresponding XML configuration file.

2.8.1 CCPL_do_individual_coupling_generation

- **SUBROUTINE CCPL_do_individual_coupling_generation(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API generates the coupling procedures among the import interfaces and export interfaces of the given component model. All MPI processes of the given component model are required to call this API at the same time. A component model can call this API multiple times before its model time is advanced.

2.8.2 CCPL_do_family_coupling_generation

- **SUBROUTINE CCPL_do_family_coupling_generation(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API generates the coupling procedures among the import interfaces and export interfaces of the given component model and its descendant component models that have not been disabled in the family coupling procedure generation. All MPI processes of the given component model are required to call this API at the same time. A component model can call this API multiple times before its model time is advanced.

2.8.3 CCPL_do_external_coupling_generation

- **SUBROUTINE CCPL_do_external_coupling_generation(num_comps, comps_full_names, individual_or_family, annotation)**
  - num_comps [INTEGER; IN]: The number of component models involved in the current coupling procedure generation. “num_comps” must be larger than 0.
  - comps_full_names [CHARACTER, DIMENSION(:); IN]: An array of full names of a set of component models. The array size of “comps_full_names” cannot be smaller than “num_comps”.
  - individual_or_family [INTEGER, DIMENSION(:), OPTIONAL; IN]: An array for specifying individual or family coupling procedure generation regarding to each of the corresponding component models. Its array size cannot be smaller than “num_comps”. Value of *1* means individual coupling procedure generation while value of *2* means family coupling procedure generation. When this parameter is not provided, all the corresponding component models are involved in individual coupling procedure generation.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

 This API generates the coupling procedures among the import interfaces and export interfaces of the given set of component models. All MPI processes in any given component model must call this API at the same time, with consistent parameters. A component model can call this API multiple times before its model time is advanced, for the coupling procedure generation with any component models.

2.8.4 CCPL_get_configurable_comps_full_names

- **SUBROUTINE CCPL_get_configurable_comps_full_names(comp_id, keyword, num_comps, comps_full_names, individual_or_family, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - keyword [CHARACTER; IN]: The keyword used to search the full names of a set of component models specified in the coupling connection configuration file (Section 3.5.3) of the given component model. It has a maximum length of 80 characters.
  - num_comps [INTEGER; OUT]: The number of the corresponding component models.
  - comps_full_names [CHARACTER, DIMENSION(:); OUT]: An array for outputting the full names of the corresponding component models. Its array size cannot be smaller than “num_comps”.
  - individual_or_family [INTEGER, DIMENSION(:), OPTIONAL; OUT]: An array for outputting the specification of individual or family coupling procedure generation regarding to each of the corresponding component models. Its array size cannot be smaller than “num_comps”. Value of *1* means individual coupling procedure generation while value of *2* means family coupling procedure generation. The default value (the corresponding information has not been specified in the coupling connection

configuration file) is 1.

- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API gets a set of full names of component models specified in the XML configuration file *comp_full_name*.coupling_connections.xml where *comp_full_name* means the full name of the component model that calls this API. When the set is not empty, the full name of the given component model will be returned through "num_comps" and "comps_full_names". Moreover, the array "individual_or_family" can be further returned if required. All MPI processes of the corresponding component model are required to call this API at the same time.

Table 9 APIs for restart management.

No.	API	Brief description
1	CCPL_do_restart_write_IO	Write the corresponding data into restart data files for a given component model. local variables or data of C-Coupler into restart data files
2	CCPL_start_restart_read_IO	Start the stage of restarting the model run based on the corresponding restart data files when the run type is "continue", "branch" or "hybrid"
3	CCPL_restart_read_fields_all	Read in the data of all restart field instances of the given component model from the corresponding restart data files in a "continue", "branch" or "hybrid" run
4	CCPL_restart_read_fields_interface	Read in the data of all restart field instances of a given import interface from the restart data files in a "continue", "branch" or "hybrid" run
5	CCPL_get_restart_setting	Get the restart setting (i.e., restart date, restart second, original case name and run type) from C-Coupler in a "continue", "branch" or "hybrid" run
6	CCPL_is_restart_timer_on	Check whether the implicit restart timer is on

2.9 APIs for adaptive restart management

C-Coupler includes a set of APIs (Table 9) for conveniently achieving restart capability of a coupled model:

1) There is a unique interface "CCPL_do_restart_write_IO" for producing restart data files. All field instances of each import interface will be written into the restart data fields no matter the setting of the corresponding coupling lag, and the field instances that has been declared as restart related (Section 2.6) will also be written into the restart data fields.

Similar to models, C-Coupler should also have the ability of restarting model simulation. C-Coupler provides two APIs accordingly, as listed out in the following Table 9. Currently, the restart management of C-Coupler only serves the local variables or data managed by C-Coupler. In other words, the restart capability of component model variables (including the component model fields that have been registered to C-Coupler as coupling field instances) is still required to be achieved by component models themselves. To achieve restart capability for a coupled model, all active component models should separately call the two APIs “CCPL_do_restart_write_IO” and “CCPL_start_restart_read_IO”. Besides “initial” run, C-Coupler supports the other three types of model run: “continue”, “branch” and “hybrid”, which are related to the restart capability.

2.9.1 CCPL_do_restart_write_IO

- **SUBROUTINE CCPL_do_restart_write_IO(comp_id, bypass_timer, bypass_import_fields, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - bypass_timer [LOGICAL; IN]: a mark used to specify whether to bypass the implicit restart timer when writing the data for restart capability. If “bypass_timer” is true, C-Coupler will produce the corresponding restart data files at any time when this API is called. Otherwise, C-Coupler will produce the corresponding restart data files only when the implicit restart timer is on.
  - bypass_import_fields [LOGICAL, OPTIONAL; IN]: C-Coupler will automatically write the values of the field instances of an import interface into restart data files in default. Without destroying the restart capability of the coupled model, users can disable such a capability to save the time in producing restart data files, through setting “bypass_import_fields” to true.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

  This API writes the data related to C-Coupler (including the field instances registered to C-Coupler) into restart data files for the given component model. The restart data files will be put under the data directory of the given component model (please refer to Section 4). All MPI processes of the given component model are required call this API at the same time, with consistent parameters. If “bypass_timer” is set to true, restart writing happens when this API is called; otherwise, the implicit restart timer of the given component model will control the time of restart writing, which means that restart writing will not truly happen when the implicit restart timer is not on. Please note that, C-Coupler forces all component models share the same implicit restart timer, which is specified by the configuration file “CCPL_dir/config/all/env_run.xml” (please refer to Section 3.1 for details).

2.9.2 CCPL_start_restart_read_IO

- **SUBROUTINE CCPL_start_restart_read_IO(comp_id, specified_restart_file, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - specified_restart_file [CHARACTER, OPTIONAL; IN]: a specific file used for reading in restart data.

- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API will read in a part of restart information from the default or an explicitly specified restart data file, and then start the stage of restart reading for the given component model, when the run type is “continue”, “branch” or “hybrid” (will be neglected when the run type is “initial”). When “specified_restart_file” is not specified, the implicitly default restart files will be determined by the type of model run (“initial”, “continue”, “branch” and “hybrid”; please refer to Section 3.1 for some details). The restart data files should have already been put under the corresponding data directories (please refer to Section 4). All MPI processes of the given component model are required to call this API at the same time. It can be called after the API “CPL_set_normal_time_step” has been called. It can be called only one time and cannot be called when the API “CCPL_advance_time” has been called.

2.9.3 CCPL_restart_read_fields_all

- **SUBROUTINE CCPL_restart_read_fields_all(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API reads in the data of all restart field instances of the given component model from the corresponding restart data files in a “continue”, “branch” or “hybrid” run, while will be neglected in an “initial” run. All MPI processes of the given component model are required to call this API at the same time. It can be called multiple times, cannot be called before calling the API “CCPL_start_restart_read_IO”, and cannot be called when “CCPL_advance_time” for the given component model has been called.

2.9.4 CCPL_restart_read_fields_interface

- **SUBROUTINE CCPL_restart_read_fields_all(interface_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given import interface.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API reads in the data of all restart field instances of the given import interface from the corresponding restart data files in a “continue”, “branch” or “hybrid” run, while will be neglected in an “initial” run. All MPI processes of the given import interface are required to call this API at the same time. It can be called multiple times, while cannot be called before calling the API “CCPL_start_restart_read_IO”, and cannot be called when “CCPL_advance_time” for the given import interface has been called.

2.9.5 CCPL_get_restart_setting

- **CCPL_get_restart_setting(comp_id, restart_date, restart_second, original_case_name, run_type, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - restart_date [INTEGER; OUT]: Used to return the restart date (YYYYMMDD) in a "continue", "branch" or "hybrid" run. -1 will be returned in an "initial" run.
  - restart_second [INTEGER; OUT]: Used to return the restart second in a "continue", "branch" or "hybrid" run. -1 will be returned in an "initial" run.
  - original_case_name [CHARACTER, OPTIONAL; OUT]: Used to return the original case name corresponding to the restart data files.
  - run_type [CHARACTER, OPTIONAL; OUT]: Used to return the run type ("initial", "continue", "branch" or "hybrid") of the current simulation run.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the restart setting (i.e., restart date, restart second, original case name and run type) in a "continue", "branch" or "hybrid" run. It can be called after the given component model has been registered.

2.9.6 CCPL_is_restart_timer_on

- **logical FUNCTION CCPL_is_restart_timer_on(comp_id, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API checks whether the implicit restart timer of the given component model is on at the current time step.

2.10 APIs for data input/output

C-Coupler3 includes a common parallel I/O framework CIOFC for helping a component model to flexibly achieve data input and output in parallel. It can adaptively input data fields from a time-series dataset during the model integration, automatically perform data interpolation when necessary (e.g., spatial interpolation when a field has different grids in the model and in the data files, and time interpolation when inputting a field from a time series dataset), and output fields either periodically or irregularly. It also allows convenient specification by users of the I/O settings; e.g., the data fields for I/O, the time series of the data files for I/O, and the data grids in the files (called file grids hereafter), based on its APIs and XML configuration files (Section 3.6).

An active component model can register an input/output handle for a set of field instances. The keyword of an input/output handler can be expressed as <ID of the component model, handler

name>. Therefore, a component model can have multiple input/output handlers with different names, while the input/output handlers of different component models can have the same name. The API for executing a handler can be called at each time step, while the handler will truly work only when its sampling timer is on at the current model time or has been bypassed. An output handler can output fields explicitly via an API call, while can also work implicitly when the component model uses C-Coupler to advance the current model time. For the detailed implementation of CIOFC, please refer to Yu et al., 2022.

Table 10 lists out the APIs for data input/output as well as their brief descriptions.

Table 10 APIs for the common parallel I/O framework CIOFC

No.	API	Brief description
1	CCPL_register_configurable_output_handler	Register an output handler whose I/O settings are specified in an XML configuration file
2	CCPL_register_normal _output_handler	Register an output handler whose I/O settings are specified via the API arguments
3	CCPL_handle_normal_explicit_output	Explicitly execute an output handler
4	CCPL_readin_field_from_dataFile	Input a field from a specific data file
5	CCPL_register_input_handler	Registering an input handler whose I/O settings are specified in an XML configuration file. It works for a set of fields and can input data values from a time-series dataset
6	CCPL_execute_input_handler	Execute an input handler

2.10.1 CCPL_register_configurable_output_handler

- **integer FUNCTION CCPL_register_configurable_output_handler (handler_name, num_field_instances, field_instance_ids, configuration_name, implicit_or_explicit, output_grid_id, handler_output_h2d_grid_id, handler_output_v1d_grid_id, sampling_timer_id, annotation)**
  - return value [INTEGER; OUT]: The ID of the new output handler.
  - handler_name [CHARACTER; IN]: The name of the new output handler. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘_’.
  - num_field_instances [INTEGER; IN]: The number of the field instances (>0) that can be outputted via this handler.
  - field_instance_ids [INTEGER, DIMENSION (:); IN]: The IDs of the field instances that can be outputted via this handler. The array size of “field_instance_ids” cannot be smaller than “num_field_instances”. All field instances must correspond to the same component model, and any two field instances cannot share the same field name. The field instances must cover all fields specified in the I/O settings in the corresponding XML configuration file (Section 3.6.1).
  - configuration_name [CHARACTER; IN]: The keyword for finding the corresponding

XML configuration file.

- implicit_or_explicit [LOGICAL; IN]: The mark for specifying whether the execution of the output handler is explicit or implicit. The value of true means implicit execution, while false means explicit execution. For implicit execution, the data output is automatically performed when advancing the model time by C-Coupler. For explicit execution, the data output is performed only when calling the corresponding API for the output handler.
- output_grid_id [INTEGER, OPTIONAL; IN]: The ID of the default file grid. A field instance will be outputted to the default grid (as well as its subgrids) when there is no configuration of its file grid. A field instance will be outputted to the model grid when no file grid is specified.
- handler_output_h2d_grid_id [INTEGER, OPTIONAL; IN]: The ID of a 2-D grid that can be referred as a file grid specified in the XML configuration file (via the special grid name "handler_output_H2D_grid" in the configuration file).
- handler_output_v1d_grid_id [INTEGER, OPTIONAL; IN]: The ID of a vertical grid (1-D) that can be referred as a file grid specified in the XML configuration file (via the special grid name "handler_output_V1D_grid" in the configuration file).
- sampling_timer_id [INTEGER, OPTIONAL; IN]: The ID of a timer that specifies a period when the output handler truly works. When it is not specified, the output handler should be executed at each time step.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API registers an output handler for a set of field instances with the I/O settings specified in an XML configuration file. It returns the ID of the new output handler when the registration succeeds. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters.

2.10.2 CCPL_register_normal_output_handler

• **integer FUNCTION CCPL_register_normal_output_handler (handler_name, num_field_instances, field_instance_ids, file_name, file_type, implicit_or_explicit, sampling_timer_id, output_timer_id, file_timer_id, output_grid_id, inst_or_aver, float_datatype, integer_datatype, annotation)**

- return value [INTEGER; OUT]: The ID of the new output handler.
- handler_name [CHARACTER; IN]: The name of the new output handler. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
- num_field_instances [INTEGER; IN]: The number of field instances (>0) output via this output handler.
- field_instance_ids [INTEGER, DIMENSION (:); IN]: The ID of the field instances that are outputted by this handler. All field instances specified by "field_instance_ids" must correspond to the same component model. The array size of "field_instance_ids" cannot

be smaller than “num_field_instances”. Any two field instances cannot share the same field name.

- file_name [CHARACTER; IN]: The name of the files that the fields will be outputted into. It must be a common file name with a unique wildcard “*”. The handler can generate new output data files under the period determined by the parameter “file_timer_id”, where the name of a new file is generated through replacing the unique wildcard “*” by the current model time.
- file_type [CHARACTER; IN]: The type of the output data files. C-Coupler3 currently only supports NetCDF (“netcdf”) type.
- implicit_or_explicit [LOGICAL; IN]: The mark for specifying whether the execution of the output handler is explicit or implicit. The value of true means implicit execution, while false means explicit execution. For implicit execution, the data output is automatically performed when advancing the model time by C-Coupler. For explicit execution, the data output is performed only when calling the corresponding API for the output handler.
- sampling_timer_id [INTEGER, OPTIONAL; IN]: The ID of a timer that specifies a period when the output handler truly works. When it is not specified, the output handler should be executed at each time step.
- output_timer_id [INTEGER, OPTIONAL; IN]: The ID of a timer that specifies the period of data output, i.e., a period when field values are written into the output data files. The period determined by “output_timer_id” should be no more frequent than that determined by “sampling_timer_id”. When the “output_timer_id” is not specified, the period of data output is determined by the specification of “sampling_timer_id”.
- file_timer_id [INTEGER, OPTIONAL; IN]: The ID of a timer that specifies a period for creating new output data files. When it is not specified, such a period is determined by the specification of “output_timer_id”.
- output_grid_id [INTEGER, OPTIONAL; IN]: The ID of the default file grid. If it is not specified, each field instance will be outputted to the model grid. Otherwise, each field instance will be outputted to the default grid as well as its subgrids.
- inst_or_aver [INTEGER, OPTIONAL; IN]: The mark for specifying outputting instantaneous or time-averaged values of all field instances. The value of 0 means outputting instantaneous values, while the value of 1 means outputting time-averaged values. The handler will output instantaneous values when “inst_or_aver” is not specified.
- float_datatype [CHARACTER, OPTIONAL; IN]: The data type of float values in output data files. Its value can be “real4” or “real8”. When it is not specified, a field in output data files will keep the same float data type as the model.
- integer_datatype [CHARACTER, OPTIONAL; IN]: The data type of integer values in output data files. Its value can be “short”, “integer” or “long”. When it is not specified, a field in output data files will keep the same integer data type as the model.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API registers an output handler whose I/O settings are fully specified via the API arguments. It returns the ID of the new output handler when the registration succeeds. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters.

2.10.3 CCPL_handle_normal_explicit_output

- **logical FUNCTION CCPL_handle_normal_explicit_output (handler_id, bypass_timer, annotation)**
  - return value [LOGICAL; OUT]: True will be returned constantly in C-Coupler3.
  - handler_id [INTEGER, IN]: The ID of an output handler that has been registered.
  - bypass_timer [LOGICAL, OPTIONAL; IN]: A mark for specifying whether to bypass the timer when executing the output handler. If “bypass_timer” has been set to true, the output handler will truly work, no matter whether the parameter sampling_timer_id for registering the output handler has been specified, and the specified output time series will also be neglected.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API executes an explicit output handler, and returns true constantly. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. When “bypass_timer” is set to true, the instantaneous values of the corresponding field instances will be outputted to a new data file when the API is called. The specification of “bypass_timer” must keep the same throughout multiple executions of the same output handler. Existing data values in a file will be overwritten when executing the same output handler multiple times at the same model time.

2.10.4 CCPL_readin_field_from_dataFile

- **logical FUNCTION CCPL_readin_field_from_dataFile (field_instance_id, data_file_name, file_type, field_name_in_file, necessity, grid_id_for_file, value_min_bound, value_max_bound, annotation)**
  - return value [LOGICAL; OUT]: If the data input from the file has been executed successfully, true will be returned; otherwise, false will be returned.
  - field_instance_id [INTEGER; IN]: The ID of the field instance to be inputted.
  - data_file_name [CHARACTER; IN]: Name of the data file that the field values will be read from.
  - file_type [CHARACTER; IN]: The type of the input data file. C-Coupler3 currently only supports NetCDF (“netcdf”) type.
  - field_name_in_file [CHARACTER, OPTIONAL; IN]: The field name in the input data file. When “field_name_in_file” is not specified, it means that the field has the same name in both the model and the input data file.

- necessity [INTEGER, OPTIONAL; IN]: a mark (value of 0 or 1) for specifying whether it is necessary to successfully input data values from the file. If its value is 0, failure in data input (e.g., the data file does not exist, or the data file does not contain the corresponding field) is allowed. Otherwise, the model run will be stopped when failure happens.
- grid_id_for_file [INTEGER, OPTIONAL; IN]: the ID of the file grid of the field. When "grid_id_for_file" is not specified, it means that the field is on the same grid in both the model and the input data file.
- value_min_bound [REAL, OPTIONAL; IN]: The lower bound of all data values inputted from the data. Error will be reported when "value_min_bound" is specified and some data values are out of bound.
- value_max_bound [REAL, OPTIONAL; IN]: The upper bound of all data values inputted from the data. Error will be reported when "value_max_bound" is specified and some data values are out of bound.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API explicitly inputs the values of a field from a specific data file. All MPI processes of the given component model are required to call this API at the same time, with consistent parameters.

2.10.5 CCPL_register_input_handler

• **integer FUNCTION CCPL_register_input_handler(handler_name, field_instances_ids, num_fields, configuration_name, input_timer_id, necessity, field_connected_status, annotation)**

- return value [INTEGER; OUT]: The ID of the new input handler that does not use an XML configuration file.
- handler_name [CHARACTER; IN]: The name of the new output handler. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
- field_instance_ids [INTEGER, DIMENSION (:); IN]: The ID of the field instances that are inputted by this handler. All field instances specified by "field_instance_ids" must correspond to the same component model. The array size of "field_instance_ids" cannot be smaller than "num_fields". Any two field instances cannot share the same field name.
- num_fields [INTEGER; IN]: The number of the field instances (>0) that can be inputted via this handler.
- configuration_name [CHARACTER; IN]: The keyword for finding the corresponding XML configuration file.
- input_timer_id [INTEGER; IN]: The ID of a timer that specifies the period of handling data output. "input_timer_id" can be specified as -1, which means the input handler will be executed whenever it be called.
- necessity [INTEGER, DIMENSION (:), OPTIONAL; IN]: an array of marks (value of 0 or 1) each of which specifies whether it is necessary to successfully input data values of

the corresponding field from the data file. If its value is 0, failure in the data input of the corresponding field (e.g., the data file does not exist, or the data file does not include the field) is allowed. Otherwise, the model run will be stopped when failure happens. The array size of “necessity” should be no smaller than “num_fields”.

- field_connected_status [INTEGER, DIMENSION (:), OPTIONAL; OUT]: The array each element of which corresponds to a field instance specified by “field_instance_ids” and returns the status (value 0 or 1) that whether there are file data values corresponding to the field instance. Status value of 1 means that there are the corresponding file data values. The array size of “field_instance_ids” should be no smaller than “num_fields”.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API registers an input handler based on an XML configuration file, and returns the ID of the new input handler when the registration succeeds. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. An input handler can adaptively input data fields from a time-series dataset specified via a configuration file, and automatically perform data interpolation (e.g., spatial interpolation and time interpolation) when necessary.

2.10.6 CCPL_execute_input_handler

• **logical FUNCTION CCPL_execute_input_handler(handler_id, annotation)**
  - return value [LOGICAL; OUT]: True will be returned constantly for C-Coupler3.
  - handler_id [INTEGER, IN]: The ID of an input handler.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API executes the specified input handler, and returns true constantly. All MPI processes of the corresponding component model are required to call this API at the same time, for the same input hander. The input handler truly works only when no “input_timer_id” is specified when registering the input handler or the input timer is on at the current model time. When the current model time is different from any time records in the dataset, time interpolation with the linear remapping algorithm will be used.

2.11 APIs for halo exchanges

The development of a component model generally includes parallelizing the serial program for effectively utilizing a large number of processor cores to accelerate model integration. A typical approach of model parallelization is to decompose the whole grid domain into a number of subdomains, assign each subdomain to a process and then make different processes conduct integration for different subdomains. To achieve correct parallelization that should keep almost exactly the same simulation results with the original serial program, processes in a parallel execution should work cooperatively, e.g., data exchanges among processes should be performed when required.

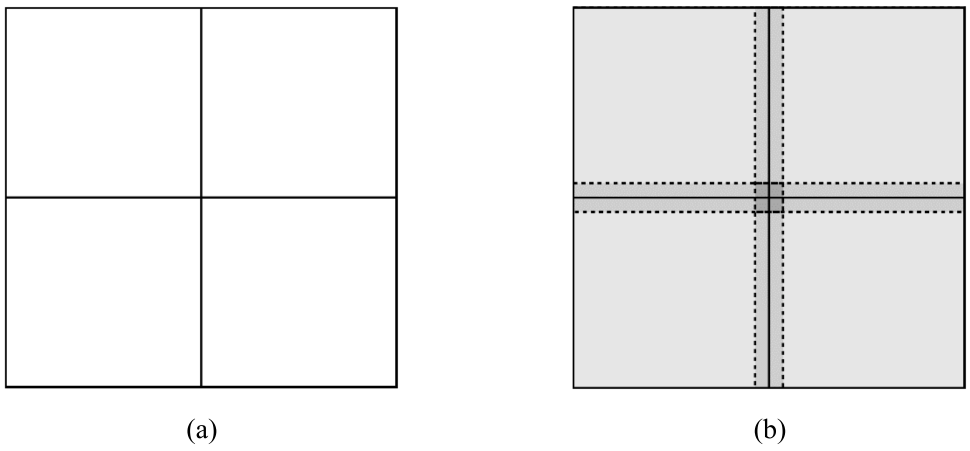

(a) (b)

Figure 2 A example of a parallel decomposition (a) and halo regions (b)

Table 11 APIs for common component parallel communication framework

No.	API	Brief description
1	CCPL_register_halo_region	Register halo regions under a parallel decomposition.
2	CCPL_register_halo_exchange_interface	Register a halo exchange interface for a set of fields.
3	CCPL_execute_halo_exchange_using_id	Execute a halo exchange interface specified via the interface ID.
4	CCPL_execute_halo_exchange_using_name	Execute a halo exchange interface specified via the interface name.
5	CCPL_finish_halo_exchange_using_id	Finish a halo exchange interface specified via the interface ID.
6	CCPL_finish_halo_exchange_using_name	Finish a halo exchange interface specified via the interface name.
7	CCPL_report_one_field_instance_checksum	Report the checksum of a field instance throughout the processes of the corresponding component model.
8	CCPL_report_all_field_instances_checksum	Report the checksums of all registered field instances of a component model

Given a component model running on 4 processes, Figure 2(a) shows a parallel decomposition of the horizontal grid, where each subdomain is assigned to each process. As there are generally data dependencies among grid cells, i.e., the integration calculation on one grid cell depends on the result values on other grid cells around it, a correct parallelization requires a subdomain in a process to be enlarged with a certain halo region for keeping the result values from other processes, and requires processes to cooperatively perform data exchanges to fill the result values in the halo regions (called halo exchange for short). Corresponding to Figure 2(a), Figure 2(b) shows the halo region in each process and the source processes of the result values in each halo region.

Therefore, it is a fundamental task for parallelizing a component model to develop a halo exchange functionality. For helping the parallelization of various component models, C-Coupler3 includes a common halo-exchange library that can work for any model grid, any parallel decomposition and any halo region. Specifically, the halo-exchange library can work for a set of fields at the same time and supports both synchronous and asynchronous communication in halo exchanges.

Table 11 lists out the APIs of halo exchange, as well as their brief descriptions.

2.11.1 CCPL_register_halo_region

- **integer FUNCTION CCPL_register_halo_region(halo_name, decomp_id, num_local_cells, local_cells_local_indexes, local_cells_global_indexes, annotation)**
  - return value [INTEGER; OUT]: The ID of the new halo region.
  - halo_name [CHARACTER; IN]: The name of the halo region. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'. There cannot be two halo regions with the same name under the same parallel decomposition.
  - decomp_id [INTEGER; IN]: The ID of an existing parallel decomposition corresponding to the halo region. In Each process, the local sub domains corresponding to the given parallel decomposition should contain all grid cells in the halo regions. When registering a parallel decomposition (Section 2.5), the global index of each cell in halo regions should be set to CCPL_NULL_INT.
  - num_local_cells [INTEGER; IN]: The number of local grid cells (≥0) in the halo region in the current MPI process.
  - local_cells_local_indexes [INTEGER; DIMENSION(:); IN]: The local index of each grid cell in the halo region of the current MPI process. The local index means the index of a grid cell in the local sub domains of current MPI process corresponding to the given parallel decomposition. The local index starts from 1. The array size of "local_cells_local_indexes" cannot be smaller than "num_local_cells".
  - local_cells_global_indexes [INTEGER; DIMENSION(:); IN]: The global index of each grid cell in the halo region of the current MPI process. The array size of "local_cells_global_indexes" cannot be smaller than "num_local_cells". Each element of "local_cells_global_indexes" should be between 1 and size of the whole grid domain.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API registers halo regions on the given parallel decomposition, and returns the ID of the new halo regions when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters.

2.11.2 CCPL_register_halo_exchange_interface

- **integer FUNCTION CCPL_register_halo_exchange_interface(interface_name, num_field_instances, field_instance_IDs, halo_region_IDs, annotation)**

- return value [INTEGER; OUT]: The ID of the new halo exchange interface.
- interface_name [CHARACTER; IN]: The name of the halo exchange interface. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'. There cannot be two halo exchange interfaces with the same name in the same component model.
- num_field_instances [INTEGER; IN]: The number of field instances (>0) exchanged via this halo exchange interface.
- field_instance_IDs [INTEGER, DIMENSION(:); IN]: The ID of the field instances that are exchanged by this interface. All field instances specified by "field_instance_IDs" must correspond to the same component model. The array size of "field_instance_IDs" cannot be smaller than "num_field_instances". Any two field instances cannot share the same field name.
- halo_region_IDs [INTEGER, DIMENSION(:); IN]: The ID of the halo regions corresponding to the given field instances (each ID corresponds to a field instance). The array size of the parameter "halo_region_IDs" cannot be smaller than the parameter "num_field_instances". All halo regions and all field instances must correspond to the same component model.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API registers a new halo exchange interface for a set of field instances, and returns the ID of the new halo exchange interface when the registration succeeds. All MPI processes of the component model are required to call this API at the same time, with consistent parameters.

2.11.3 CCPL_execute_halo_exchange_using_id

• **SUBROUTINE CCPL_execute_halo_exchange_using_id(interface_id, is_asynchronous, annotation)**
- interface_id [INTEGER; IN]: The ID of a halo exchange interface.
- is_asynchronous [LOGICAL; IN]: A mark of the asynchronous or synchronous mode for executing the interface. When "is_asynchronous" is set to *true*, the interface will be returned immediately without waiting for completement of the halo exchange. When "is_asynchronous" is set to *false*, the interface will not be returned until the halo exchange finishes.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

• **Description of this API**

This API executes a halo exchange interface based on the given interface ID. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters.

2.11.4 CCPL_execute_halo_exchange_using_name

• **SUBROUTINE CCPL_execute_halo_exchange_using_name(comp_id, interface_name, is_asynchronous, annotation)**

- comp_id [INTEGER; IN]: The ID of the component model corresponding to this interface.
- interface_name [CHARACTER; IN]: The name of the halo exchange interface to be executed. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
- is_asynchronous [LOGICAL; IN]: A mark of the asynchronous or synchronous mode for executing the interface. When "is_asynchronous" is set to *true*, the interface will be returned immediately without waiting for completement of the halo exchange. When "is_asynchronous" is set to *false*, the interface will not be returned until the halo exchange finishes.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API executes a halo exchange interface based on the given component model ID and interface name. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters.

2.11.5 CCPL_finish_halo_exchange_using_id

- **SUBROUTINE CPL_finish_halo_exchange_using_id(interface_id, annotation)**
  - interface_id [INTEGER; IN]: The ID of a given halo exchange interface.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API finishes execution of a halo exchange interface based on the given interface ID. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. If the halo exchange interface is executed asynchronously, this API will wait until the halo exchange finishes; otherwise, this API will return immediately.

2.11.6 CCPL_finish_halo_exchange_using_name

- **SUBROUTINE CCPL_finish_halo_exchange_using_name(comp_id, interface_name, annotation)**
  - comp_id [INTEGER; IN]: The ID of the component model corresponding to this interface.
  - interface_name [CHARACTER; IN]: The name of the halo exchange interface to be executed. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API finishes execution of a halo exchange interface based on the given component model ID and interface name. All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters. If the halo exchange interface is executed asynchronously, this API will wait until the halo exchange finishes; otherwise, this API will return immediately.

2.11.7 CCPL_report_one_field_instance_checksum

- **SUBROUTINE CCPL_report_one_field_instance_checksum(field_instance_id, bypass_decomp, hint, annotation)**
  - field_interface_id [INTEGER; IN]: The ID of a given field instance.
  - bypass_decomp [LOGICAL; IN]: A mark to specify whether to consider the parallel decomposition when calculating the checksum. When its value is “false”, the parallel decomposition of the field will be considered.
  - hint [CHARACTER; IN]: The hint used to mark the code position when calling this API. It has a maximum length of 512 characters. It will be outputted when reporting the checksum value.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API calculates and reports the checksum of a field instance throughout the processes of the corresponding component model. The information of checksums will be outputted to C-Coupler’s log files under the directory “CCPL_dir/run/CCPL_logs” (Section 4). All MPI processes of the corresponding component model are required to call this API at the same time, with consistent parameters.

2.11.8 CCPL_report_all_field_instances_checksum

- **SUBROUTINE CCPL_report_all_field_instances_checksum (comp_id, bypass_decomp, hint, annotation)**
  - comp_id [INTEGER; IN]: The ID of a the given component model.
  - bypass_decomp [LOGICAL; IN]: A mark to specify whether to consider the parallel decomposition when calculating the checksum. When its value is “false”, the parallel decomposition of the field will be considered.
  - hint [CHARACTER; IN]: The hint used to mark the code position when calling this API. It has a maximum length of 512 characters. It will be outputted when reporting the checksum value.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API calculates and reports the checksum of all registered field instances of the given component model throughout the processes of the component model. The information of checksums will be outputted to C-Coupler’s log files under the directory “CCPL_dir/run/CCPL_logs” (Section 4). All MPI processes of the given component model are required to call this API at the same time, with consistent parameters.

2.12 APIs for the external module integration

Table 12 APIs for external module integration.

No.	API	Brief description
1	CCPL_external_modules_inst_init	Initialize an external module instance
2	CCPL_external_modules_inst_run	Execute an external module instance
3	CCPL_external_modules_inst_finalize	Finalize an external module instance
4	CCPL_external_modules_get_local_comm	Get the MPI communicator of the host model
5	CCPL_external_modules_get_comp_ID	Get the ID of the host model
6	CCPL_external_modules_is_active_process	Check whether current MPI process is active to the external module
7	CCPL_external_modules_para_get_field_grid_ID	Get the grid ID of a field
8	CCPL_external_modules_para_get_field_decomp_ID	Get the parallel decomposition ID of a field
9	CCPL_external_modules_para_get_num_control_vars	Get the number of control variables
10	CCPL_external_modules_para_get_control_var	Get the control variable
11	CCPL_external_modules_para_get_num_timers	Get the number of timers
12	CCPL_external_modules_para_get_timer_ID	Get the ID of a timer
13	CCPL_external_modules_para_get_num_fields	Get the number of field instances
14	CCPL_external_modules_para_get_field_ID	Get the ID of a field instance
21	CCPL_external_modules_declare_argument	Declare a field
22	CCPL_external_modules_para_get_field_pointer	Get the data pointer of a field

C-Coupler3 includes a Common Module Integration Framework that works as a middle layer that enables a model to flexibly integrate and utilize an external module, e.g., a physical parameterization scheme, a flux calculation algorithm, a data assimilation algorithm, etc. This integration framework can keep the independence between a model and an external module, i.e., the external module can have its own grids, parallel decompositions and data structures of fields. The codes of an external module should be compiled into a DLL that is loaded when the model launches the module, which enables the external module to have its own compilation system.

To integrate an external module, users should develop the driving procedures, i.e., an initialization driving subroutine, an execution driving subroutine, and a finalization driving subroutine. The initialization driving subroutine is used for declaring the arguments and driving the initialization of the external module. It only has one input argument that is an ID for labelling the current external module (called module ID). The execution driving subroutine is used for driving the execution of the external module. It has two input arguments, i.e., the module ID and a chunk index that can be used for single-column parameterization schemes. The finalization driving subroutine is used for driving the finalization of the external module. It does not have any arguments. Each external module has a unique name that determines the subroutine name of each driving procedures. For example, given an external module named "EM", its three driving procedures should be named "EM_*CCPL_INIT*", "EM_*CCPL_RUN*" and "EM_*CCPL_FINALIZE*" respectively. A set of APIs are provided for the initialization driving subroutine to obtain information (e.g., grids and parallel decompositions) from the model and declare arguments of the external module, as shown in Table 12.

A model can start, run and finalize an instance of an external module, based on a set of APIs corresponding to the driving procedures and a XML configuration file format, as shown in Table 12 and Figure 21 in Section 3.7.

2.12.1 CCPL_external_modules_inst_init

- **integer FUNCTION CCPL_external_modules_inst_init (host_comp_id, inst_name, modules_name, ptype, dl_name, process_active, control_vars, field_inst_ids, timer_ids, annotation)**
  - return value [INTEGER; OUT]: The ID of the instance of external modules.
  - host_comp_id [INTEGER; IN]: The ID of the host component model that calls the API.
  - inst_name [CHARACTER; IN]: The name also the keyword of the external module instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - modules_name [CHARACTER; IN]: The name of the external modules, which specifies the information in the corresponding XML configuration file. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - ptype [CHARACTER; IN]: The type of the external module, including "individual" and "package". The type "individual" means a unique procedure, while the type "package" means a package that contains multiple procedures specified through the corresponding configuration files. When it is a unique procedure, "modules_name" must be consistent with the corresponding driving subroutines in the dynamically linked library.
  - dl_name [CHARACTER, OPTIONAL; IN]: The name of the dynamic library that contains the unique external module. It should be specified when the value of "ptype" is "individual". It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - process_active [LOGICAL, OPTIONAL; IN]: A mark for specifying whether current MPI process is active to run and finalize the external module. All processes of the current component model will call the initialization driving subroutine of each external module while only the active processes will call the running driving subroutine of each external module. When "process_active" is not specified, it means that the current process is active.
  - control_vars [INTEGER, DIMENSION(:), OPTIONAL; IN]: An integer array of control variables that can be obtained by the external module via the corresponding APIs.
  - field_inst_ids [INTEGER, DIMENSION(:), OPTIONAL; IN]: The IDs of model field instances corresponding to the arguments of each external module, e.g., these field instances should cover all input and output arguments of each external module. All these field instances should correspond to the same component model.
  - timer_ids [INTEGER, DIMENSION(:), OPTIONAL; IN]: The IDs of timers that can be

obtained by the external modules.
- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API initializes an external module instance and returns its ID. All MPI processes of the host component model are required to call this API at the same time, with the consistent parameters. This API will load in the corresponding DLLs and call the initialization driving subroutine of the corresponding external modules.

2.12.2 CCPL_external_modules_inst_run

- **SUBROUTINE CCPL_external_modules_inst_run (instance_id, chunk_index, annotation)**
  - instance_id [INTEGER; IN]: The ID of an external module instance.
  - chunk_index [INTEGER; IN]: The index of the given chunk corresponding to single-column data structures. If the corresponding external modules do not use single-column data structures, the chunk index should be -1. The index of the first chunk is 1.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API execute an external module instance, where the running driving subroutine of each corresponding external module will be called. All MPI processes of the host component model are required to call this API at the same time, with the same input parameters.

2.12.3 CCPL_external_modules_inst_finalize

- **SUBROUTINE CCPL_external_modules_inst_run (instance_id, annotation)**
  - instance_id [INTEGER; IN]: The ID of an external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API finalize an external module instance, where the finalization driving subroutine of each corresponding external module will be called. All MPI processes of the host component model are required to call this API at the same time, with the same input parameters.

2.12.4 CCPL_external_modules_get_local_comm

- **integer FUNCTION CCPL_external_modules_get_local_comm (instance_id, annotation)**
  - return value [INTEGER; OUT]: The MPI communicator of the active processes for running the given external module instance.
  - instance_id [INTEGER; IN]: The ID of an external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

This API gets the MPI communicator of the active processes for running the external module instance corresponding to the given instance ID. This API can be called at all stages of an external module.

2.12.5 CCPL_external_modules_get_comp_ID

- **integer FUNCTION CCPL_external_modules_get_comp_ID (instance_id, annotation)**
  - return value [INTEGER; OUT]: The ID of the component model that calls the external module instance.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the ID of the component model that calls the external module specified via the instance ID. This API can be called at all stages of an external module.

2.12.6 CCPL_external_modules_is_active_process

- **logical FUNCTION CCPL_external_modules_is_active_process (instance_id, annotation)**
  - return value [LOGICAL; OUT]: If current MPI process is active to run and finalize the external module, “true” will be returned; otherwise “false” will be returned.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

This API checks whether current MPI process is active to run and finalize the external module. This API can be called at all stages of an external module.

2.12.7 CCPL_external_modules_para_get_field_grid_ID

- **integer FUNCTION CCPL_external_modules_para_get_field_grid_ID (instance_id, field_name, annotation)**
  - return value [INTEGER; OUT]: The grid ID of the corresponding model field instance.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - field_name [CHARACTER; IN]: The name of the model field instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the grid ID of the model field instance corresponding to the given field name, which has been registered to C-Coupler in the component model of calling the external module. This API can be called at all stages of an external module.

2.12.8 CCPL_external_modules_para_get_field_decomp_ID

- **integer FUNCTION CCPL_external_modules_para_get_field_decomp_ID (instance_id, field_name, annotation)**
  - return value [INTEGER; OUT]: The parallel decomposition ID of the corresponding model field instance.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - field_name [CHARACTER; IN]: The name of the model field instance. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the parallel decomposition ID of the model field instance corresponding to the given field name, which has been registered to C-Coupler in the host component model of the external module. This API can be called at all stages of an external module.

2.12.9 CCPL_external_modules_para_get_num_control_vars

- **integer FUNCTION CCPL_external_modules_para_get_num_control_vars (instance_id, annotation)**
  - return value [INTEGER; OUT]: The number of control variables provided by the component model.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

  This API returns the number of control variables provided by the component model. This API can be called at all stages of an external module.

2.12.10 CCPL_external_modules_para_get_control_var

- **integer FUNCTION CCPL_external_modules_para_get_control_var (instance_id, control_var_index, annotation)**
  - return value [INTEGER; OUT]: The value of the specified control variable.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - control_var_index [INTEGER; IN]: The index of the control variable in the array specified by the parameter "control_vars" of the API CCPL_external_modules_inst_init.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

  This API returns the ID of a control variable of the component model specified when initializing the external module via calling the API CCPL_external_modules_inst_init. The number of the index should be between 1 and size of the corresponding array. This API can be called at all stages of an external module.

2.12.11 CCPL_external_modules_para_get_num_timers

- **integer FUNCTION CCPL_external_modules_para_get_num_timers (instance_id, annotation)**
  - return value [INTEGER; OUT]: The number of the timers provided by the component model.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

  This API returns the number of timers provided by the host component model. This API can be called at all stages of an external module.

2.12.12 CCPL_external_modules_para_get_timer_ID

- **integer FUNCTION CCPL_external_modules_para_get_timer_ID (instance_id, timer_index, annotation)**
  - return value [INTEGER; OUT]: The ID of the specified timer.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - timer_index [INTEGER; IN]: The index of the timer in the array specified by the parameter "timer_ids" of the API CCPL_external_modules_inst_init.

- annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the ID of a timer of the component model specified when initializing the external module via calling the API CCPL_external_modules_inst_init. The number of the index should be between 1 and size of the corresponding array. This API can be called at all stages of an external module.

2.12.13 CCPL_external_modules_para_get_num_fields

- **integer FUNCTION CCPL_external_modules_para_get_num_fields (instance_id, annotation)**
  - return value [INTEGER; OUT]: The number of field instances provided by the host model.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the number of field instances provided by the host component model. This API can be called at all stages of an external module.

2.12.14 CCPL_external_modules_para_get_field_ID

- **integer FUNCTION CCPL_external_modules_para_get_field_ID (instance_id, field_index, annotation)**
  - return value [INTEGER; OUT]: The ID of the field instance.
  - instance_id [INTEGER; IN]: The ID of the external module instance.
  - field_index [INTEGER; IN]: The index of the model field instances in the array specified by the parameter "field_inst_ids" of the API CCPL_external_modules_inst_init.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the ID of a field instance of the component model specified when initializing the external module via calling the API CCPL_external_modules_inst_init. The number of the index should be between 1 and size of the corresponding array. This API can be called at all stages of an external module.

2.12.15 CCPL_external_modules_declare_argument

- **INTEGER FUNCTION CCPL_external_modules_declare_argument (proc_inst_id, data_pointer, field_name, type_inout, decomp_id, grid_id, dims_size, field_unit, annotation)**
  - return value [INTEGER; OUT]: The ID of declared argument.
  - proc_inst_id [INTEGER; IN]: The ID of the external module instance.
  - data_pointer [REAL or INTEGER, no DIMENSION or DIMENSION|(:), (:,:), (:,:,:) or (:,:,:,:)|; INOUT]: The data buffer pointer corresponding to the declared argument. The number of dimensions of the data buffer pointer cannot be larger than the array size of "dims_size".
  - field_name [CHARACTER; IN]: The name of the declared argument. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'. A "field_name" is legal only when there is a corresponding entry in the configuration file "public_field_attribute.xml".
  - type_inout [INTEGER; IN]: The mark for specifying whether the declared argument of the external module instance is an input or output argument. The value of *"CCPL_PARA_TYPE_IN"* means an input argument passed from the component model to the external module. The value of *"CCPL_PARA_TYPE_OUT"* means an output argument passed from the external module to the component model. The value of *"CCPL_PARA_TYPE_ INOUT"* means the declared argument is both input and output.
  - decomp_id [INTEGER; IN]: The ID of the corresponding parallel decomposition. If the declared argument is a scalar variable or a field instance on a grid without a horizontal sub grid (for example, the field instance is only on a vertical grid), "decomp_id" should be specified to *-1*.
  - grid_id [INTEGER; IN]: The ID of the corresponding grid. If the declared argument is a scalar variable, "grid_id" should be set to *-1*. When both "grid_id" and "decomp_id" are not -1, the must correspond to the same component model, and the horizontal grid corresponding to "decomp_id" must be a sub grid of the grid corresponding to "grid_id".
  - dim_size [INTEGER, DIMENSION(:), OPTIONAL; IN]: An array each element of which specifies the size of a dimension of the data buffer pointer. The array size of "dim_size" must be no smaller than the number of dimensions of "data_pointer". "dim_size" should be provided when "data_pointer" points to an empty space (does not point to an existing memory space).
  - field_unit [CHARACTER, OPTIONAL; IN]: The unit of the declared argument. Default unit specified in the configuration file "public_field_attribute.xml" (Please refer to Section 3.2 for details) will be used when "field_unit" is not specified when calling this API.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

- **Description of this API**

This API declares an argument of the corresponding external module instance, and returns the ID of the new argument when the declaration succeeds. "field_name", "decomp_id", and "grid_id" are keywords of the declared argument. If the "data_pointer" points to an existing memory space, the declared argument will use this memory space, and the size of the memory space must be the same as the required size; Otherwise, "data_pointer" will be assigned to point to a memory space managed by C-Coupler, and such memory space will be shaped following the dimension sizes specified by the "dim_size". All MPI processes of the corresponding component model should call this API for starting the declaration of an argument, while more calls of this API will be required when the argument has multiple chunks in the current MPI process (each call of this API declares a chunk of the argument).

2.12.16 CCPL_external_modules_get_argument_pointer

- **SUBROUTINE CCPL_external_modules_get_argument_pointer (proc_inst_id, field_name, data_pointer, chunk_index, annotation)**
  - proc_inst_id [INTEGER; IN]: The ID of the external module instance.
  - field_name [CHARACTER; IN]: The name of the corresponding argument that has been declared. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'. Different field instances can share the same "field_name". A "field_name" is legal only when there is a corresponding entry in the configuration file "public_field_attribute.xml".
  - data_pointer [REAL or INTEGER; no DIMENSION or DIMENSION|(:), (:,:), (:,:,:) or (:,:,:,:)|; INOUT]: The data pointer pointing to the returned memory space corresponding to the given argument. The number of dimensions of the data pointer should be same as the corresponding dimension number when declaring the corresponding argument.
  - chunk_index [INTEGER; IN]: The index of chunk in the declared argument. If the declared argument includes multiple chunks, "chunk_index" should be specified properly; otherwise, any value of "chunk_index" will be ignored. The index of the first chunk is 1.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API is used to get a data buffer pointer from an argument of an external module instance. The number of dimensions of "data_pointer" should be same as the corresponding dimension number when declaring the corresponding argument. If the declared argument includes multiple chunks, "chunk_index" should be specified properly.

2.13 APIs for ensemble data assimilation

Data assimilation (DA) provides better initial states of model runs by combining observational information and models. Ensemble-based DA methods that depend on the ensemble run of a model have been widely used. In response to the development of seamless prediction based on coupled models or even earth system models, coupled DA is now in the mainstream of DA development. In addition to the scientific challenges of DA methods, there are technical challenges in developing a coupled ensemble DA system, which have not yet been satisfactorily addressed. These include how to conveniently (1) achieve an ensemble run of a coupled model, (2) integrate the software of a coupled model and the software of ensemble DA methods into a robust system, and (3) achieve efficient interaction between the ensemble of the coupled model and the DA methods, especially as model resolution improves.

To fully overcome the technical challenges, a Data Assimilation Framework based on C-Coupler (DAFCC) was designed and developed. DAFCC that currently supports weakly coupled ensemble DA becomes a part of C-Coupler, which enables users to conveniently integrate a DA method into a model as a procedure that can be directly called by the model. DAFCC can automatically and efficiently handle online data exchanges between the model ensemble members and the DA method, and enable the DA method to utilize more processor cores in parallel execution.

Specifically, the development of DAFCC is based on the common module integration framework. So, each DA algorithm should become an external module enclosed in a DLL with the driving procedures, and the online data exchanges are achieved by the common module integration framework.

A model ensemble can start, run and finalize an instance of a DA algorithm, based on a set of APIs and a XML configuration file format, as shown in Table 13 and Figure 22 in Section 3.8.

Table 13 APIs for the data assimilation framework.

No.	API	Brief description
1	CCPL_ensemble_modules_inst_init	Initialize a DA algorithm instance
2	CCPL_ensemble_modules_inst_run	Execute a DA algorithm instance
3	CCPL_external_modules_para_get_ensemble_size	Get the number of ensemble members
4	CCPL_external_modules_para_get_ensemble_member_index	Get the index of the ensemble member running on current process

2.13.1 CCPL_ensemble_modules_inst_init

- **integer FUNCTION CCPL_ensemble_modules_inst_init (member_comp_id, inst_name, field_inst_ids, control_vars, annotation)**
  - return value [INTEGER; OUT]: The ID of the DA algorithm instance.
  - member_comp_id [INTEGER; IN]: The ID of the component model that runs the DA algorithm.
  - inst_name [CHARACTER; IN]: The name of the DA algorithm instance, which is the keyword of the DA procedure instance and also specifies the corresponding configuration information in the XML configuration file. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
  - field_inst_ids [INTEGER, DIMENSION(:); IN]: The list of IDs of model field instances. It has the same meaning with the parameter "field_inst_ids" of the API CCPL_external_modules_inst_init.
  - control_vars [INTEGER, DIMENSION(:); IN]: An integer array of control variables. It has the same meaning with the parameter "control_vars" of the API

CCPL_external_modules_inst_init.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API initializes a DA algorithm instance and returns the ID of the DA algorithm instance when succeeds. All MPI processes of the model ensemble are required to call this API at the same time, with consistent parameters. The main flowchart of this API is as follows:

1) Determine the host model of the DA algorithm according to the corresponding information in the configuration file. If the DA algorithm is an individual algorithm that operates on the data of each ensemble member separately, each ensemble member will be a host model. Otherwise, the host model will be the ensemble-set component model that will be generated automatically.

2) Prepare information from the host model, such as the model grids, parallel decompositions, and field instances, which the initialization driving subroutine of the DA algorithm can obtain via the corresponding APIs.

3) Initialize the corresponding DA algorithm according to the corresponding algorithm name and DLL name specified in the corresponding configuration file, where the corresponding DLL will be linked to the host model and the corresponding initialization driving subroutine in the DLL will be called. This implementation enables the user to conveniently change the DA algorithms used in simulations via the configuration file without modifying the code of the model.

4) Set up data exchange operations according to the input or output arguments of the DA algorithm instance declared in the initialization driving subroutine via the corresponding APIs. The data exchange is divided into two levels: the data exchange between the ensemble members and DAFCC, and the data exchange between DAFCC and the DA algorithm. DAFCC enables a DA algorithm instance to use instantaneous model results or time statistical results (i.e., mean, maximum, cumulative, and minimum) in a time window, and also enables an ensemble DA algorithm instance to use aggregated results or statistical results (ensemble-mean, ensemble-anomaly, ensemble-maximum, or ensemble-minimum) from ensemble members.

2.13.2 CCPL_ensemble_modules_inst_run

- **SUBROUTINE CCPL_ensemble_modules_inst_run (instance_id, annotation)**
  - instance_id [INTEGER; IN]: The ID of the DA algorithm instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API executes a DA algorithm instance. All MPI processes of the model ensemble are required to call this API at the same time. Each DA algorithm has a timer specified via the configuration file, which determines when the DA algorithm actually runs, e.g., this API can be called for the DA algorithm instance at each time step, while it actually runs only when the corresponding timer is on. The input and output data of a DA algorithm can be put under the working directory of the corresponding component model, or under a specific directory specified in the configuration file. The main flowchart of this API is as follows:

1) Execute the data exchange operations for the input fields of the DA algorithm. This step automatically transfers the input fields from each ensemble member of the corresponding component model to DAFCC and then from DAFCC to the DA algorithm, where the statistical processing regarding the time window or the ensemble is done at the same time.

2) Execute the DA algorithm instance through calling the running driving subroutine of the DA algorithm.

3) Execute the data exchange operations for the output fields of the DA algorithm. This step automatically transfers the output fields from the DA algorithm to DAFCC and then from DAFCC to each ensemble member of the corresponding component model.

Table 14 APIs for parallel debugging.

No.	API	Brief description
1	CCPL_report_log	Write a log for a given component model into a corresponding log file managed by C-Coupler.
2	CCPL_report_progress	Write a progress report for a given component model into a corresponding log file managed by C-Coupler.
3	CCPL_report_error	Write an error report for a given component model into a corresponding log file managed by C-Coupler and then stop the whole model run.
4	CCPL_get_comp_log_file_name	Get the log file name of the current process of the given component model. This log file is not used by C-Coupler, which means that C-Coupler will not write any report into this log file.
5	CCPL_get_comp_log_file_device	Get the device ID of the log file of the current process of the given component model. If the log file has not been opened, it will be opened to a device ID. This log file is not used by C-Coupler, which means that C-Coupler will not write any report into this log file.

2.13.3 CCPL_external_modules_para_get_ensemble_size

- **integer FUNCTION CCPL_external_modules_para_get_ensemble_size (external_inst_id, annotation)**
  - return value [INTEGER; OUT]: The size of ensemble members.
  - external_inst_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.

- **Description of this API**

This API returns the size of ensemble members when the external module is used in an ensemble data assimilation.

2.13.4 CCPL_external_modules_para_get_ensemble_member_index

- **integer FUNCTION CCPL_external_modules_para_get_ensemble_member_index (external_inst_id, annotation)**
  - return value [INTEGER; OUT]: The ensemble member index of current process.
  - external_inst_id [INTEGER; IN]: The ID of the external module instance.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API, which is recommended but not mandatory, and should be provided by the user. It has a maximum length of 512 characters.
- **Description of this API**

This API returns the index of ensemble member running on current process, when the external module is used in an ensemble data assimilation.

2.14 APIs for parallel debugging

2.14.1 CCPL_report_log

- **SUBROUTINE CCPL_report_log(comp_id, condition, report_string, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - condition[LOGICAL; IN]: the log will be reported only when “condition” is set to *true*.
  - report_string [CHARACTER; IN]: the log to be written into the log files. It has a maximum length of 512 characters.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API reports a log for a given component model. Please refer to Section 3.3 for the details that how C-Coupler supports parallel debugging.

2.14.2 CCPL_report_progress

- **SUBROUTINE CCPL_report_progress(comp_id, condition, report_string, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - condition[LOGICAL; IN]: the progress log will be reported only when “condition” is set to *true*.
  - report_string [CHARACTER; IN]: the progress log to be written into the log files. It has a maximum length of 512 characters.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API reports a progress log for a given component model. Please refer to Section 3.3 for the details that how C-Coupler supports parallel debugging.

2.14.3 CCPL_report_error

- **SUBROUTINE CCPL_report_error(comp_id, condition, report_string, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - condition[LOGICAL; IN]: the error log will be reported and the model run will be ended only when "condition" is set to *false*.
  - report_string [CHARACTER; IN]: the error log to be written into the log files. It has a maximum length of 512 characters.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API reports an error log for a given component model. Please refer to Section 3.3 for the details that how C-Coupler supports parallel debugging.

2.14.4 CCPL_get_comp_log_file_name

- **logical FUNCTION CCPL_get_comp_log_file_name(comp_id, file_name, annotation)**
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - file_name [CHARACTER; OUT]: The log file name of the current process of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API gets the log file name (Section 4) of the current process of the given component model. This log file is put under the directory created by C-Coupler but is not used by C-Coupler, which means that C-Coupler will not write any report into this log file.

2.14.5 CCPL_get_comp_log_file_device

- **integer FUNCTION CCPL_get_comp_log_file_device(comp_id, annotation)**
  - return value [INTEGER; OUT]: The device ID of the log file of the current process of the given component model.
  - comp_id [INTEGER; IN]: The ID of the given component model.
  - annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
- **Description of this API**

This API gets the device ID of the log file (Section 4) of the current process of the given component model. This log file is put under the directory created by C-Coupler but is not used by C-Coupler, which means that C-Coupler will not write any report into this log file. If this log file has not been opened, it will be opened to a device ID.

2.15 Examples of a coupled model with C-Coupler APIs

Figure 3 shows an example of how to use the C-Coupler APIs to achieve hybrid coupling configuration and model coupling in the initialization stage of a coupled model. There are four component models, *comp1 ~ comp4*. We assume that *comp1* and *comp2* are coupled together, *comp3* and *comp4* are coupled together, *comp3* and *comp4* are the children of *comp1* and depend on some boundary conditions from *comp1*. Firstly, the two component models *comp1* and *comp2* that cover all MPI processes (process 0~34) and do not share any MPI process, call the API *CCP_register_component* at the same time, to register themselves as the root component models. *Comp3* and *comp4*, the two children of *Comp1*, partially share a subset of MPI processes (process 9~12). All MPI processes of *comp3* first register *comp3* as a child of *comp1*, and next set the unique time step, register several model grids, register a parallel decomposition, register several coupling field instances, specify a coupling field instance as the dynamic surface field of a 3-D grid, define several timers, and register several coupling interfaces. After calling the API "*CCPL_do_individual_coupling_generation*" for the coupling procedure generation intra *comp3* itself, *comp3* executes some coupling interfaces and next finalizes its coupling configuration stage through calling the API "*CCPL_end_coupling_configuration*". *comp4* follows a C-Coupler flowchart similar to *comp3*. As *comp3* and *comp4* share some processes, they cannot conduct coupling configuration and model coupling at the same time in most cases (in this example, we specifically make *comp3* earlier than *comp4*), except for the simultaneous call of the API "*CCPL_do_external_coupling_generation*" that can generate coupling procedures for the coupling connections between *comp3* and *comp4*. After both *comp3* and *comp4* finish their coupling configuration stage, their parent *comp1* conduct its coupling configuration, with a flowchart similar to *comp3* and *comp4*. As *comp1* share processes with *comp3* and *comp4*, *comp1* cannot conduct coupling registration simultaneously with *comp3* and *comp4*, and we specifically make *comp1* later than *comp3* and *comp4* in this example. As *comp2* does not share any process with *comp1*, *comp3* or *comp4*, *comp2* can conduct coupling registration simultaneously with *comp1*, *comp3* and *comp4*. Finally, *comp1* and *comp2*, the root component models, call the API "*CCPL_end_coupling_configuration*" simultaneously to finalize the coupling configuration stage of themselves and the whole coupled model, and to invoke the coupling generator to conduct a global coupling procedure generation. At the end of the initialization stage, each component model can read in the restart data files when necessary.

Figure 4 shows an example of model coupling in the kernel (time integration) stage of the coupled model referred in Figure 3, where we still use the assumption in Figure 3 and further assume that *comp1* and *comp2* have the same time step that is two times of the time step of *comp3* and *comp4*. In this example, all coupling interfaces are executed without bypassing the timers. At a time step of *comp1* and *comp2*, they can execute coupling interfaces at the same time, call the API "*CCPL_do_restart_write_IO*" to generate restart data files when the restart timer is bypassed or is on, and finally call the API "*CCPL_advance_time*" to advance the model time managed by C-Coupler. We highly propose to check the consistency of model time between a component model and C-Coupler through calling the API "*CCPL_check_current_time*". *comp3* and *comp4* alternately use a C-Coupler flowchart similar to *comp1* and *comp2*, while *comp3* and *comp4* will advance their model time twice when *comp1* and *comp2* advance their model time once.

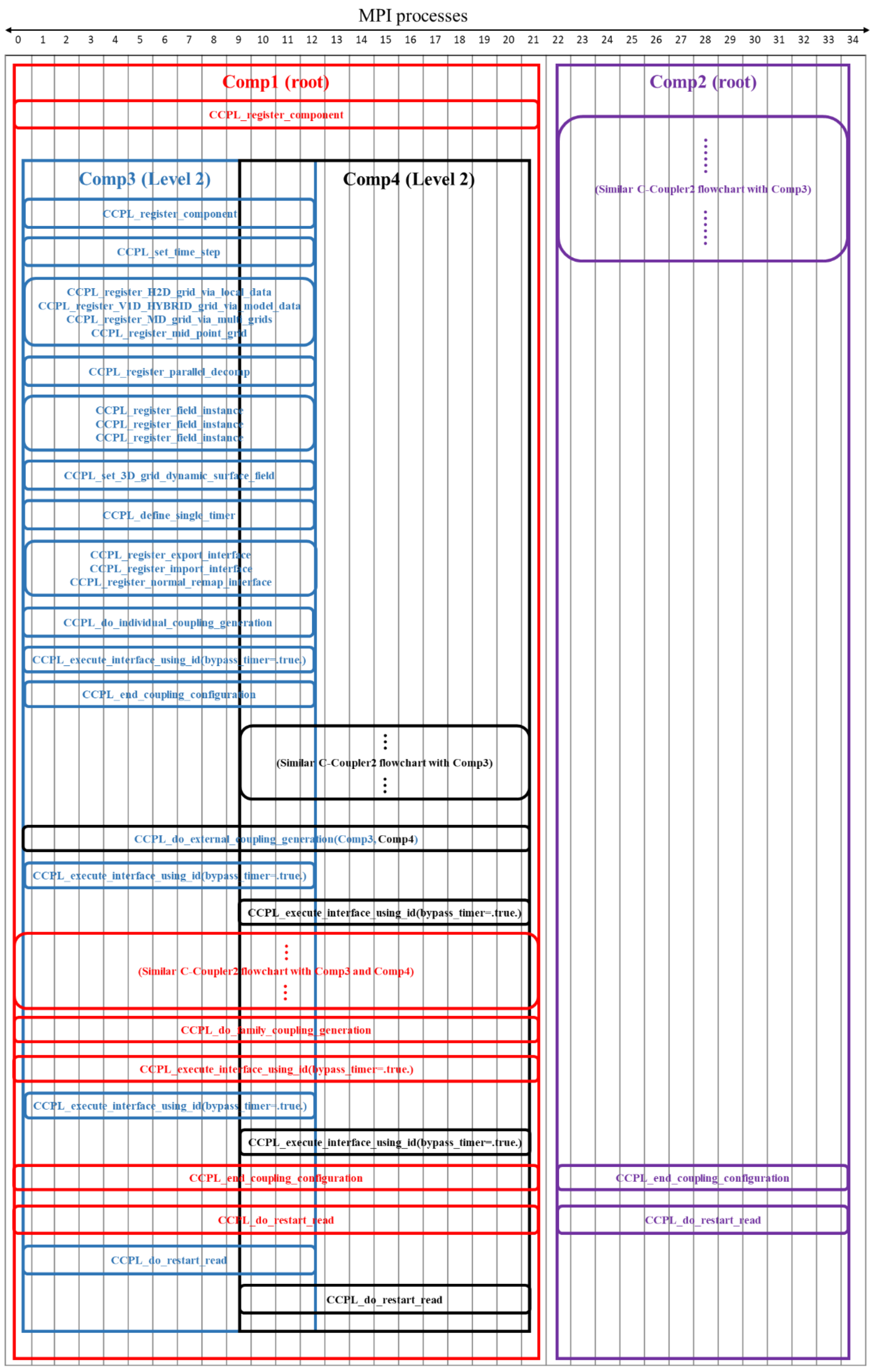

MPI processes
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
Comp1 (root)
CCPL_register_component
Comp3 (Level 2)
Comp4 (Level 2)
CCPL_register_component
CCPL_set_time_step
CCPL_register_H2D_grid_via_local_data
CCPL_register_V1D_HYBRID_grid_via_model_data
CCPL_register_MD_grid_via_multi_grids
CCPL_register_mid_point_grid
CCPL_register_parallel_decomp
CCPL_register_field_instance
CCPL_register_field_instance
CCPL_register_field_instance
CCPL_set_3D_grid_dynamic_surface_field
CCPL_define_single_timer
CCPL_register_export_interface
CCPL_register_import_interface
CCPL_register_normal_remap_interface
CCPL_do_individual_coupling_generation
CCPL_execute_interface_using_id(bypass_timer=.true.)
CCPL_end_coupling_configuration
(Similar C-Coupler2 flowchart with Comp3)
CCPL_do_external_coupling_generation(Comp3, Comp4)
CCPL_execute_interface_using_id(bypass_timer=.true.)
CCPL_execute_interface_using_id(bypass_timer=.true.)
(Similar C-Coupler2 flowchart with Comp3 and Comp4)
CCPL_do_family_coupling_generation
CCPL_execute_interface_using_id(bypass_timer=.true.)
CCPL_execute_interface_using_id(bypass_timer=.true.)
CCPL_execute_interface_using_id(bypass_timer=.true.)
CCPL_end_coupling_configuration
CCPL_do_restart_read
CCPL_do_restart_read
CCPL_do_restart_read
Comp2 (root)
(Similar C-Coupler2 flowchart with Comp3)
CCPL_end_coupling_configuration
CCPL_do_restart_read

Figure 3 An example of hybrid coupling configuration and model coupling in the initialization stage of a coupled model constructed with C-Coupler. Comp1 ~ comp4 are the four component models. The texts and boxes in the same color corresponds to the same component model.

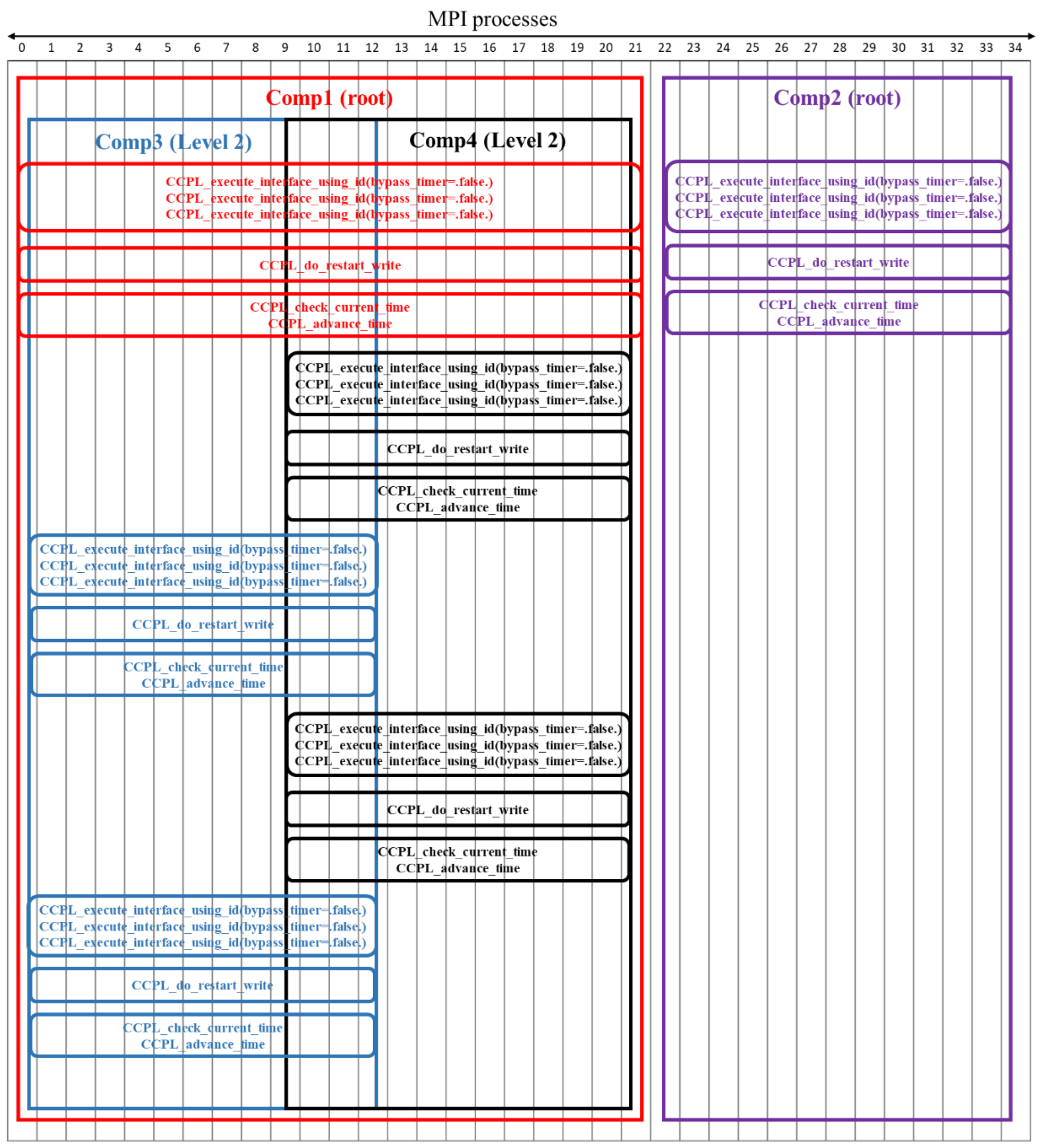
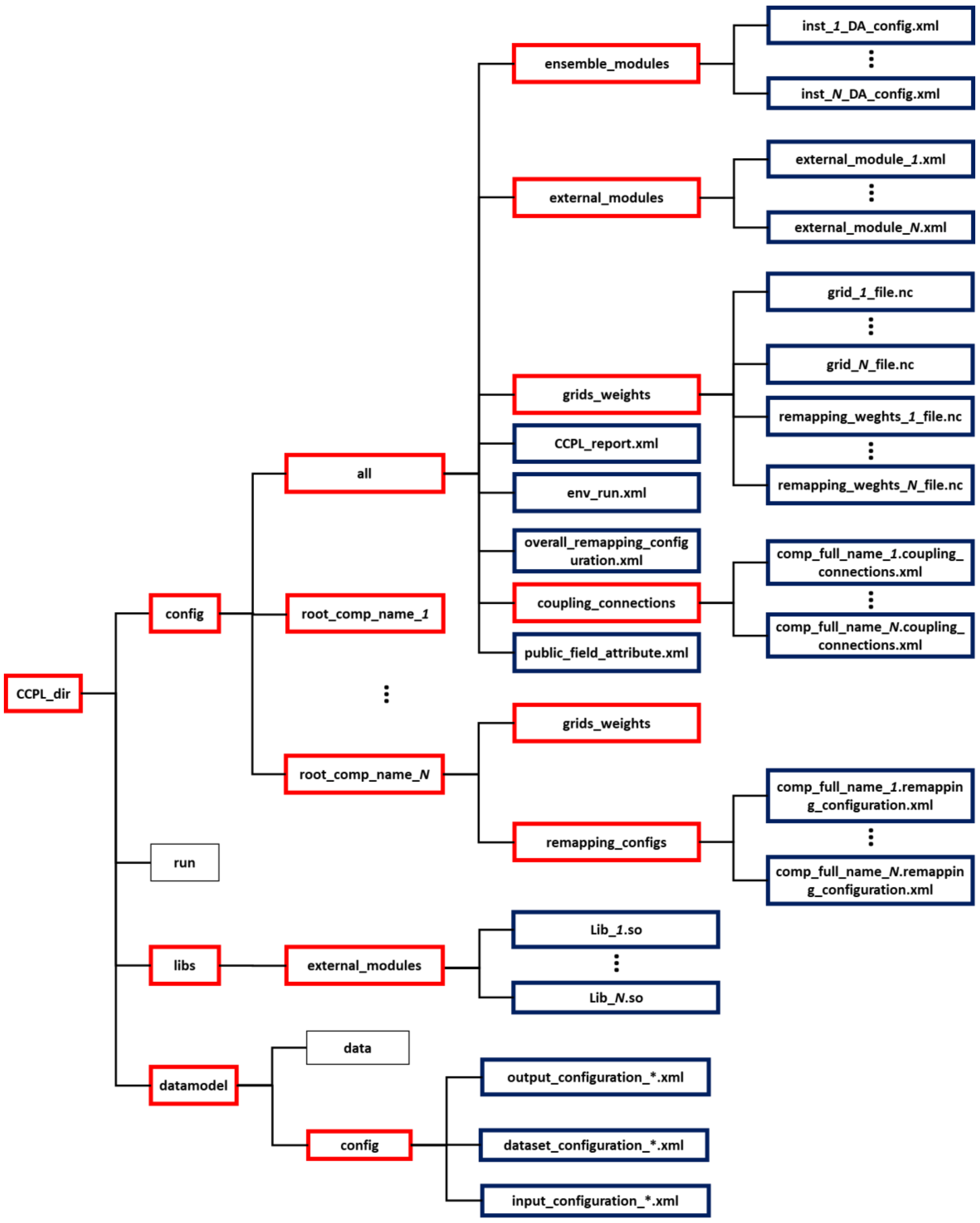


Figure 4 An example of model coupling in the kernel (time integration) stage of a coupled model constructed with C-Coupler. *Comp1 ~ comp4* are the four component models. The texts and boxes in the same color corresponds to the same component model.

3 C-Coupler configuration files



CCPL_dir
config
all
ensemble_modules
inst_1_DA_config.xml
inst_N_DA_config.xml
external_modules
external_module_1.xml
external_module_N.xml
grids_weights
grid_1_file.nc
grid_N_file.nc
remapping_weghts_1_file.nc
remapping_weghts_N_file.nc
CCPL_report.xml
env_run.xml
overall_remapping_configuration.xml
coupling_connections
comp_full_name_1.coupling_connections.xml
comp_full_name_N.coupling_connections.xml
public_field_attribute.xml
root_comp_name_1
root_comp_name_N
grids_weights
remapping_configs
comp_full_name_1.remapping_configuration.xml
comp_full_name_N.remapping_configuration.xml
run
libs
external_modules
Lib_1.so
Lib_N.so
datamodel
data
config
output_configuration_*.xml
dataset_configuration_*.xml
input_configuration_*.xml

Figure 5 Directory of the configuration files of C-Coupler. A red-edged box stands for a directory and a blue-edged box stands for a specific configuration file.

Table 15 C-Coupler Configuration files

File name	Format	Description
env_run.xml	XML	Used to specify the parameters to run a simulation of the coupled model
public_field_attribute.xml	XML	Used to specify the attributes of each coupling field
CCPL_report.xml	XML	Used to enable or disable the log report and error check by C-Coupler
overall_remapping_configuration.xml	XML	Used to specify how to remap coupling fields for the whole coupled model
comp_full_name_*N*.remapping_configuration.xml	XML	Used to specify how to remap coupling fields for a family of component models
remapping_weights_*N*_file.nc	NetCDF	A data file with remapping weights from one model grid to another
comp_full_name_*N*.coupling_connections.xml	XML	Used to specify coupling connections related to a component model
grid_*N*_file.nc	NetCDF	A data file with the grid data of a horizontal grid
inst_*N*_DA_config.xml	XML	Used to specify the configuration of a DA algorithm instance
external_module_*N*.xml	XML	Used to specify the configuration of one package of external modules
Lib_*N*.so	DLL	A dynamic link library that encloses a set of external modules
output_configuration_*.xml	XML	The configuration file corresponding to an output handler
input_configuration_*.xml	XML	The configuration file corresponding to an input handler
dataset_configuration_*.xml	XML	The configuration file for a dataset. The data files of a dataset can be put under the directory “CCPL_dir/datamodel/data”

The C-Coupler configuration files enable to flexibly specify the public coupling configuration information including shared input parameters for a model run, attributes of coupling fields, remapping configurations, and coupling connections. A configuration file is either a text file in XML format or a data file in NetCDF format. As shown in Figure 5, all configuration files are put under the directory “CCPL_dir/config”. The configuration files that are shared by the whole coupled model are put under the directory “CCPL_dir/config/all”. The configuration files specific to a given component model are put under a directory “CCPL_dir/config/root_comp_name_*N*”, where “root_comp_name_*N*” stands for the name of the root component model corresponding to the given component model. Both of the directories “CCPL_dir/config/all” and “CCPL_dir/config/root_comp_name_*N*” contain a subdirectory “grids_weights” that can store a set of grid data files for grid registration and a set of remapping weight files for remapping coupling fields. The “CCPL_dir/config/root_comp_name_*N*” also contains a subdirectory “remapping_configs” that stores a set of XML configuration files for specifying how to remap coupling fields.

Table 15 briefly describes each kind of configuration files. The grid data file “grid_*N*_file.nc” has already been introduced in section 2.4.3. In the following context of this chapter, each of remaining configuration files will be introduced. As “overall_remapping_configuration.xml”, “comp_full_name_*N*.remapping_configuration.xml” and “remapping_weights_*N*_file.nc” are used for describing the configuration of remapping, they will be introduced together.

3.1 env_run.xml

```
<?xml version="1.0" ?>
<Time_setting
      case_name="C-Coupler testing"
      model_name="ideal_model_for_CCPL2"
      run_type="initial"
      leap_year="on"
      start_date="00040331"
      start_second="0"
      rest_freq_unit="seconds"
      rest_freq_count="14400"
      rest_ref_case="C-Coupler testing"
      rest_ref_date="00040401"
      rest_ref_second="0"
      stop_option="nhours"
      stop_date="00010101"
      stop_second="0"
      stop_n="30"
/>
```

Figure 6 An example of the configuration file “env_run.xml”

As shown in Figure 6, the configuration file “env_run.xml” enables users to specify a series of global attributes for the simulation run of the whole coupled model. These global attributes are described as follows:

- case_name [CHARACTER]: The name of the simulation run. It has a maximum length of 80 characters.
- case_description [CHARACTER, OPTIONAL]: The description of the simulation run. It has a maximum length of 1000 characters.
- model_name [CHARACTER]: The name of the whole coupled model. It has a maximum length of 80 characters.
- run_type [CHARACTER]: The type to run the simulation. Four types are supported by C-Coupler: initial run, continue run, branch run and hybrid run. “run_type” must be set to “initial”, “continue”, “branch” or “hybrid” correspondingly. Please note that, in a continue or branch run, when the stop option is not “date”, the restarted time will be treated as the

start time for calculating the stop time. For example, given an initial run that stops at the model time 19900101-00000, its continue run with the "stop_option" of "ndays" and with the "stop_n" of "5", the continue run will stop at the model time 19900106-00000.

- leap_year [CHARACTER]: To specify whether leap year is enabled in the simulation run. "leap_year" must be set to "on" or "off", where "on" means that leap year is enabled.
- start_date [INTEGER]: The date to start the simulation run. It must be a positive value in the format of YYYYMMDD, where YYYY means year, MM means month and DD means day.
- start_second [INTEGER]: The second of start time on the start day. It must be no smaller than 0 and smaller than 86400.
- rest_freq_unit [CHARACTER]: The unit of the period to produce restart data files. The unit can be second ("second", "seconds", "nsecond" or "nseconds"), day ("day", "days", "nday" or "ndays"), month ("month", "months", "nmonth" or "nmonths") or year ("year", "years", "nyear" or "years"). It also can be "none" which means that no restart data files will be produced in the simulation run.
- rest_freq_count [INTEGER, OPTIONAL]: The count of the period of producing restart data files corresponding to the unit. When "rest_freq_unit" is not set to "none", "rest_freq_count" must be set to a positive value.
- rest_ref_case [CHARACTER, OPTIONAL]: When the "run_type" has been set to "branch" or "hybrid", "rest_ref_case" should be set to the name of a reference simulation case whose restart data files will be used for restarting the current simulation run.
- rest_ref_date [INTEGER]: A date that is used to specify a detailed restart data file produced by the reference simulation case. It must be a positive value in the format of YYYYMMDD, where YYYY means year, MM means month and DD means day. It must be set when the "run_type" has been set to "branch" or "hybrid".
- rest_ref_second [INTEGER]: A second number that is used to specify a detailed restart data file produced by the reference simulation case. It must be no smaller than 0 and smaller than 86400. It must be set when the "run_type" has been set to "branch" or "hybrid".
- stop_option [CHARACTER]: The option for specifying how to stop the simulation run. It can be set to date ("date"), or a time unit such as second ("second", "seconds", "nsecond" or "nseconds"), minute ("minute", "minutes", "nmunite" or "nmunites"), hour ("hour", "hours", "nhour" or "nhours"), day ("day", "days", "nday" or "ndays"), month ("month", "months", "nmonth", "nmonths") and year ("year", "years", "nyear" or "nyears").
- stop_date [INTEGER, OPTIONAL]: The model date to stop the simulation run. It must be a positive value in the format of YYYYMMDD, where YYYY means year, MM means month and DD means day. It must be set when the "stop_option" has been set to "date".
- stop_second [INTEGER, OPTIONAL]: The second number on the "stop_date" to stop the simulation run. It must be no smaller than 0 and smaller than 86400. It must be set when the "stop_option" has been set to "date".
- stop_n [INTEGER]: A number that is used to control the time length when the "stop_option" is set to a time unit. In other words, it must be set when the "stop_option" has been set to a time unit. It can be a positive value or -999. When it is a negative value -*999*, it indicates that the simulation run is endless.

3.2 public_field_attribute.xml

```
<?xml version="1.0" ?>
<field name="evap" dimensions="H2D" long_name="water evaporation" type="flux" default_unit="kg/s/m^2" />
<field name="atm_t" dimensions="V3D" long_name="air temperature" type="state" default_unit="kelvin" />
<field name="ps" dimensions="H2D" long_name="surface pressure" type="state" default_unit="Pa" />
<field name="ifrac" dimensions="H2D" long_name="fraction of sea ice" type="state" default_unit="unitless" />
<field name="sst" dimensions="H2D" long_name="sea surface temperature" type="state" default_unit="kelvin" />
<field name="sss" dimensions="H2D" long_name="salty surface temperature" type="state" default_unit="kelvin" />
<field name="tbot" dimensions="H2D" long_name="bottom atm level temperature" type="state" default_unit="kelvin" />
<field name="CO2_avg" dimensions="0D" long_name="averaged CO2 concentration" type="state" default_unit="parts per million (ppm)"/>
```

Figure 7 An example of the configuration file "public_field_attribute.xml"

As introduced in Section 2.6, when registering a filed instance that will be involved in model coupling, its field name should have a corresponding entry in the configuration file "public_field_attribute.xml" that is shared by all component models in a coupled model. When the coupling generator tries to automatically generate coupling procedures, field names are used to analyze possible coupling connections between coupling interfaces: an import interface and an export interface can have a coupling connection only when their coupling field instances have common field names.

Figure 7 show an example of the configuration file "public_field_attribute.xml" where each XML node of "field" specifies a set of attributes corresponding to a distinct field name. These attributes are described as follows:

- name [CHARACTER]: The field name. It has a maximum length of 80 characters. Each character must be 'A'~'Z', 'a'~'z', 0~9 or '_'.
- dimensions [CHARACTER]: A label of grid dimensions corresponding to the field. It can be "0D", "H2D", "V1D", "V3D". "0D" means that the field is a scalar variable not on any grid. "H2D" means that the field is on a horizontal grid. "V1D" means that the field is on a vertical grid. "V3D" means that the field is on a 3-D grid that consists of a horizontal grid and vertical grid.
- long_name [CHARACTER]: The description of the field. It has a maximum length of 1000 characters.
- Type [CHARACTER]: The type of the field: "state" or "flux".
- Default_unit [CHARACTER]: The default unit of the field. It has a maximum length of 80 characters.

3.3 CCPL_report.xml

```
<?xml version="1.0" ?>
<Report_setting
        report_internal_log="off"
        report_external_log="off"
        report_error="off"
        report_progress="on"
        flush_log_file="on"
        output_H2D_grid="on"
/>
```

Figure 8 An example of the configuration file “public_field_attribute.xml”

C-Coupler can report a lot of log information for itself and component models, can perform a lot of automatic error check, and enables each process of the coupled model to have its own log file, which of course can facilitate parallel debugging and improve the reliability in constructing a coupled model. Considering that log information and automatic error checks should be necessary for constructing a coupled model but would be time-consuming and redundant for a coupled model that is stable enough for usage after a lot of software testing, C-Coupler enables users to enable/disable the log information and error checks through the configuration file “public_field_attribute.xml”. Figure 8 shows an example of this configuration file. The attributes in this file are described as follows:

- report_internal_log [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disable (the value is “off”) C-Coupler to report its own log information. Its default value is “off”, which means that C-Coupler will not report its own log information when the configuration file does not include this attribute.
- report_external_log [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disable (the value is “off”) C-Coupler to report log information of component models. Its default value is “off”, which means that C-Coupler will not report the log information of component models when the configuration file does not include this attribute.
- report_error [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disable (the value is “off”) C-Coupler to perform time-consuming error check. Its default value is “off”, which means that C-Coupler will not perform time-consuming error check when the configuration file does not include this attribute.
- report_progress [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disable (the value is “off”) C-Coupler to report execution progress. Its default value is “off”, which means that C-Coupler will not report execution progress when the configuration file does not include this attribute. Please note that, only the root process (ID is 0) of a component model can report execution progress.
- flush_log_file [CHARACTER, OPTIONAL]: It is used to specify whether to enable to buffer logs (the value is “off”) or to force to flush the log file (the value is “on”) when trying to output a log to the log file. Its default value is “off”, which means that logs will

be buffered before writing them into a log file.

- output_H2D_grid [CHARACTER, OPTIONAL]: It is used to specify whether to output the horizontal grids to the NetCDF files under the directory "CCPL_dir/run/data/all/internal_H2D_grids". Its default value is "off", which means no output.

3.4 Remapping configuration

Remapping configuration files enable to flexibly and conveniently specify how to remap coupling fields between grids:

1) For the remapping from a source horizontal grid to a target horizontal grid, users can use the remapping weights that are either automatically generated by C-Coupler in parallel, or read from an existing remapping weight file produced by an external software tool such as SCRIP, ESMF, YAC, CoR, etc. To check whether a remapping weight file is used by C-Coupler, the user can refer to the progress report (please refer to Section 3.3 for how to enable progress report), with the key word "remapping weight file".
2) Similar to C-Coupler1, C-Coupler still uses the "2-D + 1-D" manner to achieve 3-D remapping. Regarding a 3-D remapping, the remapping configuration for the 2-D (horizontal) remapping and for the 1-D (vertical) remapping can be specified separately, where the 2-D remapping can also use the remapping weights loaded from a remapping weight file.
3) Different coupling fields in the same component model can have different remapping configurations, and the same coupling field in different component models can have different remapping configurations.
4) Given a coupling field, a component model can use its own remapping configuration, or use the remapping configuration inherited from its parent (if its own remapping configuration is not specified); for a root component model that does not have the parent, when its own remapping configuration is not specified, it can use the specified overall remapping configuration or use the default remapping configuration set by C-Coupler (if the overall remapping configuration is not specified). In the default remapping configuration, the bilinear remapping algorithm is used for remapping the "*state*" fields between horizontal grids, the conservative remapping algorithm is used for remapping the "*flux*" fields between horizontal grids, and the linear remapping algorithm is used for remapping in both the vertical dimension and the time dimension. Please note that all remapping weights in the default remapping configuration are generated by C-Coupler automatically.
5) A remapping configuration file consists of a set of remapping settings, each of which can specify the remapping configuration for all coupling fields, the coupling fields with the same type ("*flux*" or "*state*"), or a set of specific coupling fields (or even only one field). A priority strategy is designed accordingly: a remapping setting corresponding to all coupling fields is at the lowest priority, a remapping setting corresponding to a type of coupling fields is at the medium priority and a remapping setting corresponding to specific coupling fields is at the highest priority.

Given a coupling field on a coupling connection between two different component models, the procedure of data remapping will be generated for this coupling field when the corresponding grids in the two component models are different. It is possible that the remapping configuration of this coupling field is not the same in the two component models. For such a case, C-Coupler will only use the remapping configuration in the source component model (the component model that exports the coupling field). In general speaking, given a coupling field on a coupling connection, C-Coupler only uses the remapping configuration in the source component model for coupling procedure generation. Therefore, it is meaningless to specify remapping configurations for the imported coupling fields of a component model.

In the following context of this subsection, we will further introduce the remapping weight file and the configuration file.

3.4.1 Remapping weight file

The remapping weight file should be in a NetCDF format oriented from the remapping weight file produced by the software SCRIP. The software such as ESMF and CoR can also produce such format of remapping weight files. Figure 9 shows an example of this format, with the variables necessary to C-Coupler. A remapping weight file with these variables can be used by C-Coupler, no matter which software generates the remapping weight file. These variables are briefly described in Table 16.

Table 16 Descriptions of the variables required by C-Coupler in a remapping weight file

Variable	description
yc_a	Latitude (Y) value of the center of each cell of the source grid.
xc_a	Longitude (X) value of the center of each cell of the source grid.
yc_b	Latitude (Y) value of the center of each cell of the target grid.
xc_b	Longitude (X) value of the center of each cell of the target grid.
yv_a	Latitude (Y) value of the vertexes of each cell of the source grid. This variable is optional.
xv_a	Longitude (X) value of the vertexes of each cell of the source grid. This variable is optional.
yv_b	Latitude (Y) value of the vertexes of each cell of the target grid. This variable is optional.
xv_b	Longitude (X) value of the vertexes of each cell of the target grid. This variable is optional.
mask_a	Mask of each cell of the source grid.
mask_b	Mask of each cell of the target grid.
area_a	Area of each cell of the source grid.
area_b	Area of each cell of the target grid.
col	Global cell index in the source grid in each remapping weight.
row	Global cell index in the target grid in each remapping weight.
S	Weight value in each remapping weight.

```
dimensions:
        n_a = 70560 ;
        n_b = 7680 ;
        nv_a = 4 ;
        nv_b = 4 ;
        n_s = 84988 ;
variables:
        double yc_a(n_a) ;
                yc_a:units = "degrees" ;
        double yc_b(n_b) ;
                yc_b:units = "degrees" ;
        double xc_a(n_a) ;
                xc_a:units = "degrees" ;
        double xc_b(n_b) ;
                xc_b:units = "degrees" ;
        double yv_a(n_a, nv_a) ;
                yv_a:units = "degrees" ;
        double xv_a(n_a, nv_a) ;
                xv_a:units = "degrees" ;
        double yv_b(n_b, nv_b) ;
                yv_b:units = "degrees" ;
        double xv_b(n_b, nv_b) ;
                xv_b:units = "degrees" ;
        int mask_a(n_a) ;
                mask_a:units = "unitless" ;
        int mask_b(n_b) ;
                mask_b:units = "unitless" ;
        double area_a(n_a) ;
                area_a:units = "square radians" ;
        double area_b(n_b) ;
                area_b:units = "square radians" ;
        int col(n_s) ;
        int row(n_s) ;
        double S(n_s) ;
```

Figure 9   An example of the NETCDF format of the remapping weight file

3.4.2 Configuration file "*remapping_configuration.xml"

"*remapping_configuration.xml" is an XML formatted configuration file that enables users to specify the rules of how to remap the coupling fields (for example, Figure 10). Its unique XML node "root" contains a set of active XML nodes of "remapping_setting" (for example, L1 to L15, L16 to L27, and L28 to L39 in Figure 10), each of which specifies a rule of how to remap some coupling fields. An XML node of "remapping_setting" consists of two active XML nodes: an active XML node of “remapping_algorithms” (for example, L2 to L13, L17 to L23, and L29 to L34 in Figure 10) that is about the algorithms or weight files for remapping, and an active XML node of “fields” (for example, L14, L24 to L26, L35 to L38 in Figure 10) that is about the coupling fields the XML node of “remapping_setting” is for. An XML node of “remapping_setting” is active only when its attribute of “status” is set to “on” (for example, L1, L16, and L28 in Figure 10), and will be neglected when its attribute of “status” is set to “off”.

- **XML node of “remapping_algorithms”**

An XML node of “remapping_algorithms” can include at most one active XML node of “H2D_algorithm” (for example, L3 to L5, and L18 in Figure 10), at most one active XML node of “V1D_algorithm” (for example, L6 to L8, and L30 to L33 in Figure 10), and at most one active XML node of “H2D_weights” (for example, L9 to L12, and L19 to L22 in Figure 10). An XML node of “remapping_algorithms” is active only when its attribute of “status” is set to “on” (for example, L3, and L18 in Figure 10), and will be neglected when its attribute of “status” is set to “off”.

- ■ **XML node of “H2D_algorithm”**

An XML node of “H2D_algorithm” is used to specify a remapping algorithm for data interpolation between two horizontal grids, where the remapping algorithm is specified through the attribute of “name” (for example, L3, and L18 in Figure 10) and the algorithm parameters can be further set in this XML node (for example, L4 in Figure 10). An XML node of “H2D_algorithm” is active only when its attribute of “status” is set to “on” (for example, L3, and L18 in Figure 10), and will be neglected when its attribute of “status” is set to “off”.

- ■ **XML node of “V1D_algorithm”**

An XML node of “V1D_algorithm” is similar to an XML node of “H2D_algorithm”, but for data interpolation between two vertical grids (for example, L6 to L8, and L30 to L33 in Figure 10).

- ■ **XML node of “H2D_weights”**

An XML node of “H2D_weights” enables users to specify a set of external remapping weight files each of which is for data interpolation between two specific horizontal grids (for example, L9 to L12, and L19 to L22 in Figure 10). A remapping weight file must be a NETCDF file following the format of remapping weight file produced by the remapping software SCRIP (called SCRIP format hereafter). Other remapping software such as CoR can also produce SCRIP formatted remapping weight file. An XML node of “H2D_weights” is active only when its attribute of “status” is set to “on” (for example, L9, and L19 in Figure 10), and will be neglected when its attribute of “status” is set to “off”.

An XML node of “remapping_algorithms” can include an active XML node of “H2D_algorithm” and an active XML node of “H2D_weights” at the same time while “H2D_weights” has a higher priority than “H2D_algorithm”. When both a specified remapping weight file and the specified remapping algorithm can be used for handling the data interpolation between two specific horizontal grids, only the specified remapping weight file will be used.

- **XML node of “fields”**

XML nodes of “fields” enable users to specify different remapping settings for different subsets of coupling fields, so as to improve the flexibility of remapping configuration. In an XML node of “fields”, users can determine how to specify coupling fields, through setting the attribute of “specification” to “name” (for example, L35 in Figure 10), “type” (for example, L24 in Figure 10), or “default” (for example, L14 in Figure 10). When the attribute of “specification” is set to “name”, the name of the coupling fields must be listed out in the XML node of “fields” (for example, the field names of “t_atm_3D” and “ghs_atm_3D” at L36 and L37 in Figure 10). When the attribute of “specification” is set to “type”, only one type of coupling fields should be specified in the XML node of “fields” (for example, L25 in Figure 10). Currently, there are only two types of coupling fields: “state” and “flux”, and the type of each field is set in the configuration file “*public_field_attribute.xml*” (in Section 3.2). When the attribute of “specification” is set to “default”, no more information is required (for example, L14 in Figure 10), and it means that any coupling field can use the corresponding remapping setting.

An XML node of “fields” is active only when its attribute of “status” is set to “on” (for example, L14, L24, and L35 to L38 in Figure 10), and will be neglected when its attribute of “status” is set to “off”.

- **Constraints and rules to use “*remapping_configuration.xml”**

When using “*remapping_configuration.xml” to specify remapping configurations, users should note the following constraints and rules:

1) When multiple active XML nodes of “remapping_setting” are included, there must be no conflicts of fields setting between these XML nodes:
   a) There is at most one active XML node of “remapping_setting” where the attribute of “specification” in the corresponding XML node of “fields” is set to “default”.
   b) There can be at most two active XML nodes of “remapping_setting” where the attributes of “specification” in the corresponding XML nodes of “fields” are both set to “type” and the field types in these two XML nodes must be different, because currently C-Coupler only support two types of fields (“state” and “flux”).
   c) There can be multiple active XML nodes of “remapping_setting” where the attributes of “specification” in the corresponding XML nodes of “fields” are set to “name”, while there must be no common field name among these XML nodes.
2) Given a coupling field, multiple active XML nodes of “remapping_setting” can have different priorities in determining the specific remapping setting for the coupling field: the active XML node with the names of coupling fields specified has the highest priority; the active XML node with a type of coupling fields specified has a medium priority; the active XML node for the “default” (the corresponding attribute of “specification” is set to “default”) has the lowest priority.
3) It is possible that the remapping setting of a 3-D coupling field is determined by two active XML nodes of “remapping_setting” in the same configuration file or even in different configuration files, where one active XML node determines how to remap between horizontal grids and the other determines how to remap between vertical grids.

```
      <root>
L1:     <remapping_setting    status="on">
L2:         <remapping_algorithms status="on">
L3:             <H2D_algorithm status="on"    name="bilinear">
L4:                 <parameter name="enable_extrapolate" value="true" />
L5:             </H2D_algorithm>
L6:             <V1D_algorithm status="on" name="linear">
L7:                 <parameter name="enable_extrapolate" value="true" />
L8:             </V1D_algorithm>
L9:             <H2D_weights    status="on">
L10:                <file name="map_to_global_grid1_default.nc" />
L11:                <file name="map_to_regional_grid1_default.nc" />
L12:            </H2D_weights>
L13:        </remapping_algorithms>
L14:        <fields    status="on" specification="default" />
L15:    </remapping_setting>

L16:    <remapping_setting    status="on">
L17:        <remapping_algorithms status="on">
L18:            <H2D_algorithm    status="on"    name="conserv_2D" />
L19:            <H2D_weights    status="on">
L20:                <file name="map_to_global_grid1_conserv.nc" />
L21:                <file name="map_to_regional_grid1_conserv.nc" />
L22:            </H2D_weights>
L23:        </remapping_algorithms>
L24:        <fields    status="on" specification="type">
L25:            <entry value="flux" />
L26:        </fields>
L27:    </remapping_setting>

L28:    <remapping_setting    status="on">
L29:        <remapping_algorithms status="on">
L30:            <V1D_algorithm status="on" name="linear">
L31:                <parameter name="enable_extrapolate" value="true" />
L32:                <parameter name="use_logarithmic_coordinate" value="true" />
L33:            </V1D_algorithm>
L34:        </remapping_algorithms>
L35:        <fields    status="on" specification="name">
L36:            <entry value="t_atm_3D" />
L37:            <entry value="ghs_atm_3D" />
L38:        </fields>
L39:    </remapping_setting>
      </root>
```

Figure 10   An example about “*remapping_configuration.xml”

3.5 *Comp_full_name*.coupling_connections.xml

Comp_full_name.coupling_connections.xml is an XML formatted configuration file corresponding to the component model with the full name of *Comp_full_name*. It enables users to flexibly set the sources for the input coupling fields and input horizontal grids (for example, Figure 11). It consists of a set of XML nodes at different levels. An XML node with the attribute of "status" is active only when the value of "status" is "on", but not "off". In other words, an XML node will be neglected when its attribute "status" (if have) is "off". The root XML node of this configuration file consists of an XML node of "local_import_interfaces" that specifies the sources for the coupling fields of each import interface (registered to C-Coupler through the API "CCPL_register_import_interface") (for example, L1 to L36 in Figure 11), an XML node of "local_grids" that specifies the source for each horizontal grid that will be registered based on the horizontal grid from another component model (registered to C-Coupler through the API "CCPL_register_H2D_grid_from_another_component") (for example, L37 to L40 in Figure 11), and an XML node of "component_full_names_sets" that each set of component models that will be involved in a coupling procedure generation.

3.5.1 XML node of "local_import_interfaces"

The XML node of "local_import_interfaces" consists of a set of XML nodes of "import_interface", each of which specifies the sources for the coupling fields of an import interface (for example, L2 to L11 in Figure 11 is for the import interface "receive_from_OCN", and L12 to L35 is for the import interface "receive_from_ATM"). An XML node of "import_interface" consists of a set of XML nodes of "import_connection", each of which corresponds to a subset of coupling fields (for example, L13 to L21, L22 to L30, and L31 to L34 in Figure 11 are the XML nodes of "import_connection" for the import interface named "receive_from_ATM"). An XML node of "import_connection" consists of an active XML node of "fields" (for example, L4, L14 to L17, L23 to L26, and L32 in Figure 11) and an active XML node of "components" (for example, L5 to L7, L18 to L20, L27 to L29, and L33 in Figure 11). The XML node of "fields" specifies the coupling fields regarding to the corresponding "import_connection", and the XML node of "components" specifies the component models or the export interfaces that provide the coupling fields that are specified in the XML node of "fields".

- **XML node of "fields"**

In an XML node of "fields", there is an attribute of "default" with the value of "off", "remain" or "all" (for example, L4, L14, L23, and L32 in Figure 11). When the value is "off", the names of coupling fields are required be added into the XML node of "fields" (for example, L15 to L16, and L24 to L25 in Figure 11). When the value is "remain" or "all", no field name can be added into the XML node of "fields" (for example, L4, and L32 in Figure 11). The value of "remain" means all the remaining coupling fields that have not been referred in the previous XML nodes of "import_connection" for the same import interface. The value of "all" means all coupling fields of the corresponding import interface.

```
     <root>
L1:    <local_import_interfaces>
L2:       <import_interface   name="receive_from_OCN"   status="on">
L3:           <import_connection status="on">
L4:               <fields status="on" default="all" />
L5:               <components status="on" default="off">
L6:                   <component comp_full_name="ocn_unique" interface_name="send_data_to_CPL" />
L7:               </components>
L8:           </import_connection>
L9:           <import_connection status="off">
              …
L10:          </import_connection>
L11:      </import_interface>
L12:      <import_interface   name="receive_from_ATM"   status="on">
L13:          <import_connection status="on">
L14:              < fields status="on" default="off">
L15:                 <field   name="prec"   />
L16:                 <field   name="lwdn"   />
L17:              </fields>
L18:              < components status="on" default="off">
L19:                 <component comp_full_name="atm_global@atm_nest_1" />
L20:              </components>
L21:          </import_connection>
L22:          <import_connection status="on">
L23:              < fields status="on" default="off">
L24:                 <field   name="u"   />
L25:                 <field   name="v"   />
L26:              </fields>
L27:              < components status="on" default="off">
L28:                 <component interface_name="send_data_to_CPL" />
L29:              </components>
L30:          </import_connection>
L31:          <import_connection status="on">
L32:              < fields status="on" default="remain" />
L33:              < components status="on" default="all" />
L34:          </import_connection>
L35:      </import_interface>
L36:   </local_import_interfaces>

L37:   <local_grids>
L38:      <entry local_grid_name="atm_global_grid"   another_comp_full_name="atm_global"
another_comp_grid_name="H2D_grid" />
L39:      <entry  local_grid_name="atm_nest1_grid"   another_comp_full_name="  atm_global@atm_nest_1"
another_comp_grid_name="H2D_grid" />
L40:   </local_grids>

L41:   <component_full_names_sets>
L42:      <component_full_names_set   status="on" keyword="external_comps_for_coupling_generation" >
L43:          <entry comp_full_name="atm_global" individual_or_family="family" />
L44:          <entry comp_full_name="ocn_global" />
L45:      </component_full_names_set>
L46:   </component_full_names_sets>
    </root>
```

Figure 11   An example about Comp_full_name.coupling_connections.xml

- **XML node of “components”**

In an XML node of “components”, there is also an attribute of “default” with the value of “off” or “all” (for example, L5, L18, L27 and L33 in Figure 11). If the value is set to “off”, detailed information of the component models or the export interfaces that will provide the corresponding coupling field instances are required to be added into the XML node of “components” (for example, L5 to L7, L18 to L20, and L27 to L29 in Figure 11); otherwise, no full name of component models can be added into the XML node of “components” (for example, L33 in Figure 11). The value of “all” means any component model in the coupled model.

- **XML node of “import_interface”**

An XML node of “import_interface” can include several XML nodes of “import_connection”, each of which corresponds to a separate subset of coupling fields of the corresponding import interface. In other words, different XML nodes of “import_connection” cannot share any coupling fields. An XML node of “import_interface” can include only one XML node of “import_connection” when the attribute of “default” in the corresponding XML node of “fields” is set to “all” (for example, L2 to L11 in Figure 11). There can be several XML nodes of “import_connection” where the attribute of “default” in the corresponding XML nodes of “fields” are all set to “off”, optionally following an XML node of “import_connection” where the attribute of “default” in the corresponding XML node of “fields” is set to “remain” (for example, L13 to L34 in Figure 11).

For a brief summary, a valid XML node of “import_interface” must obey the constraint that the source of a coupling field is specified only once.

3.5.2 XML node of “local_grids”

The XML node of “local_grids” includes a set of entries each of which corresponds to a horizontal grid that will be registered through the API “CCPL_register_H2D_grid_from_another_component” (for example, L37 to L40 in Figure 11). In an entry, the attribute of “local_grid_name” specifies the name of the horizontal grid, which is the parameter “grid_name” when calling the API; the attribute of “another_comp_full_name” specifies the full name of the source component model and the attribute of “another_comp_grid_name” specifies the name of the corresponding horizontal grid registered in the source component model.

3.5.3 XML node of “component_full_names_sets”

The XML node of “component_full_names_sets” consists of a number of XML nodes of “component_full_names_set”. An XML node of “component_full_names_set” includes a set of entries each of corresponds of a component model. In an entry, the attribute of “comp_full_name” specifies the full name of the corresponding component model, and the optional attribute of “individual_or_family” can be set to “individual” or “family” (the default value is “individual”) that specifies individual or family coupling procedure generation of the corresponding component model.

3.6 Configuration files for data input/output

```
<output_configuration>
    <data_files status ="on" file_names="a6.output*.nc" time_format_in_filename="YYYY-MM-DD-SSSSS"
            file_type="NetCDF">
    </data_files>
    <horizontal_grids>
        ...
    </horizontal_grids>
    <vertical_grids>
        ...
    </vertical_grids>
    <V3D_grids>
       ...
    </V3D_grids>
    <fields_output_settings>
        <fields_output_setting status="on" ile_mid_name="F1D"
time_format_in_data_file="YYYYMMDDSSSSS" field_specification="hybrid" inst_or_aver="aver"
default_float_type="real4" default_integer_type="short">
            <time_setting status="on" file_freq_count="2" file_freq_unit="days">
                <time_period status="on" freq_count="1" freq_unit="days">
                    …
                </time_period>
            </time_setting>
            <fields>
                <field name_in_model="V" name_in_file="GMAO-3DS__V" float_type="real8"
grid_name="original_grid"/>
                <field name_in_model="EVAP" name_in_file="GMAO-2D__EVAP" grid_name="V3D_grid1"/>
                <field name_in_model="T"/>
            </fields>
        </fields_output_setting>
        <fields_output_setting status="on" file_mid_name="F2D"
time_format_in_data_file="YYYYMMDDSSSSS" field_specification="default">
            <time_setting status="on" file_freq_count="2" file_freq_unit="days" >
                <time_period status="on" freq_count="2" freq_unit="days" />
            </time_setting>
            <fields>
                <field name_in_model="U" name_in_file="GMAO-3DS__U" float_type="real8"/>
            </fields>
        </fields_output_setting>
    </fields_output_settings>
</root>
```

Figure 12 An example of an "output_configuration_*configuration_name*.xml"

3.6.1 Configuration file for data output

Each configuration file of data output is named "output_configuration_*configuration_name*.xml" and put under the directory "CCPL_dir/datamodel/config/", where configuration_name corresponds to the parameter "configuration_name" of the API CCPL_register_configurable_output_handler. It is an XML file that enables users to flexibly specify output configurations (for example, Figure 12). It has a unique root node for enclosing all configuration information (for example, L1 in Figure 12). The root node contains a set of child nodes for specifying the output data files (the XML node "data_files", e.g., L2~L4 in Figure 12), spatial grids (the XML nodes named "horizontal_grids", "vertical_grids" and "V3D_grids" respectively, e.g., L5~L13 in Figure 12) and variables to be outputted (the XML node "fields_output_settings", e.g., L14~L35 in Figure 12).

3.6.1.1 XML node "data_files"

A unique active "data_files" node is used for specifying names of the output data files (for example, L2 to L4 in Figure 12). A "data_files" node contains the following attributes:

- file_names [CHARACTER]: It is the common name of the output data files. It has a unique wildcard "*" that will be replaced with the current model time when creating a new output data file. Output data files can be put under a fixed directory if the common file name includes an absolute path (e.g., the first character in the common file name is '/' under Linux). Otherwise, output data files will be put under the working directory of the corresponding component model.
- time_format_in_filename [CHARACTER]: It is the time format that will be used for replacing the unique wildcard "*" in the common file name. Various time formats are supported, as shown in Table 17.
- file_type [CHARACTER]: It specifies the type of output data files. Currently, only NetCDF files are supported.

Table 17 Time formats currently supported in C-Coupler

Time formats	Description
YYYY	Calendar time only with year
YYYYMM, YYYY-MM	Calendar time only with year (YYYY) and month (MM)
YYYYMMDD, YYYY-MM-DD	Calendar time with year (YYYY), month (MM), and day (DD)
YYYYMMDD.SSSSS, YYYY-MM-DD.SSSSS, YYYYMMDD-SSSSS, YYYY-MM-DD-SSSSS, YYYYMMDDSSSSS	Calendar time with year (YYYY), month (MM), day (DD) and second (SSSSS)
YYYYMMDD.HHMMSS, YYYY-MM-DD.HH-MM-SS, YYYYMMDD.HH-MM-SS, YYYY-MM-DD.HHMMSS, YYYY-MM-DD-HH-MM-SS, YYYYMMDDHHMMSS	Calendar time with year (YYYY), month (MM), day (DD), hour (HH), minute (MM) and second (SS)
SSSSS	Time in a day in seconds (one day has 86,400 s)
HHMMSS, HH-MM-SS, HH:MM:SS	Time in a day with hour (HH), minute (MM), and second (SS)

MMDD.HHMMSS, MMDD-HHMMSS, MM-DD.HH-MM-SS, MM-DD-HH-MM-SS, MMDDHHMMSS	Calendar time with month (MM), day (DD), hour (HH), minute (MM) and second (SS)
MMDD.SSSSS, MMDD-SSSSS, MM-DD.SSSSS, MM-DD-SSSSS, MMDDSSSSS	Calendar time with month (MM), day (DD), and second (SSSSS)
DD.HHMMSS, DD-HHMMSS, DD.HH-MM-SS, DD-HH-MM-SS, DDHHMMSS	Calendar time with month day (DD), hour (HH), minute (MM) and second (SS)
DD.SSSSS, DD-SSSSS, DDSSSSS	Calendar time with month day (DD), and second (SS)
YYYYMMDD.HHMM, YYYY-MM-DD.HH-MM, YYYYMMDD.HH-MM, YYYY-MM-DD.HHMM, YYYY-MM-DD-HH-MM, YYYYMMDDHHMM	Calendar time with year (YYYY), month (MM), day (DD), hour (HH), and minute (MM)
HHMM, HH-MM, HH:MM	Time in a day with hour (HH), and minute (MM)
MMDD.HHMM, MMDD-HHMM, MM-DD.HH-MM, MM-DD-HH-MM, MMDDHHMM	Calendar time with month (MM), day (DD), hour (HH), and minute (MM)
DD.HHMM, DD-HHMM, DD.HH-MM, DD-HH-MM, DDHHMM	Calendar time with day (DD), hour (HH), and minute (MM)
YYYYMMDD.HH, YYYY-MM-DD.HH, YYYYMMDD.HH, YYYY-MM-DD.HH, YYYY-MM-DD-HH, YYYYMMDDHH	Calendar time with year (YYYY), month (MM), day (DD), and hour (HH)
HH	Time in a day only with hour
MMDD.HH, MMDD-HH, MM-DD.HH, MM-DD-HH, MMDDHH	Calendar time with month (MM), day (DD) and hour (HH)

3.6.1.2 XML node “fields_output_settings”

A unique XML node “fields_output_settings” is used for specifying how to output model fields (for example, L14 to L35 in Figure 12). It can include a number of active XML nodes “fields_output_setting”, each of which corresponds to group of fields that will be outputted to the same data file at a model time.

The head information of a “fields_output_setting” node (for example, L16 in Figure 12) contains the following attributes:

- file_mid_name [CHARACTER]: It is used for distinguishing the output data files corresponding to different “fields_output_setting” nodes. The value of “file_mid_name” will be used to generate the name of output data files, i.e., it is the prefix of the string for replacing the wildcard “*” in the common file name.
- time_format_in_data_file [CHARACTER]: It is a time format. The model time when outputting model fields will be written into output data files under the given time format. Various time formats are supported, as shown in Table 17.

– field_specification [CHARACTER]: It marks how to specify the fields to be outputted. Three types of specification are supported: “default”, “fields” and “hybrid”. “default” means that all fields specified via the parameter “field_instance_ids” when calling the API CCPL_register_configurable_output_handler will be outputted under the default settings. “hybrid” is similar to “default” while some fields can have their own settings specified in the unique child node “fields” of the “fields_output_setting” node. “fields” means that only the fields specified in the “fields” node will be outputted.
– inst_or_aver [CHARACTER]: It specifies outputting instantaneous (value “inst”) or time-averaged (value “aver”) values of fields in default.
– default_output_grid [CHARACTER, OPTIONAL]: It specifies the default file grid for the current group of fields. This default grid has higher priority than the default grid specified via the parameter “output_grid_id” when calling the API CCPL_register_configurable_output_handler. This default grid should already have been declared in the current configuration file.
– default_float_type [CHARACTER, OPTIONAL]: It specifies the default float data type of the current group of fields in output data files. Its value is either “float” or “double”.
– default_integer_type [CHARACTER, OPTIONAL]: It specifies the default integer data type of the current group of fields in output data files. Its value is “short”, “int”, or “long”.

A “fields_output_setting” node includes two child nodes, i.e., an active “time_setting” node for specifying the output time series (for example, L17 to L21 in Figure 12), and a “fields” node for configurating a set of fields (for example, L17 in Figure 12).

The head information of a “time_setting” node can specify the period for creating new out data files with the following attributes (for example, L17 in Figure 12 specifies a 2-day period). If the period is not specified, a new output data file will be created at each model time when writing the fields into the data file.

– file_freq_unit [CHARACTER, OPTIONAL]: The unit of the period. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”, “nyear” or “years”).
– file_freq_count [INTEGER, OPTIONAL]: The count of the period corresponding to the period unit. “file_freq_unit” and “file_freq_count” should be set or not set at the same time.

The remainder content of a “time_setting” node specifies the output time series, which will be further introduced in Section 3.6.1.3.

A “fields” node consists of a set of XML nodes of “field” (for example, L22 to L26 in Figure 12), each of which is the specific configuration of a distinct field. A “field” node includes the following attributes:

– name_in_model [CHARACTER]: The field name in the model.
– name_in_file [CHARACTER, OPTIONAL]: The field name in the output data files. When it is not specified, a field has the same name in both in the model and data files.
– float_type [CHARACTER, OPTIONAL]: The float data type of the field in output data files. When it is not specified, the default float data type or the data type of the field in the model is used. Its value is either “float” or “double”.
– integer_type [CHARACTER, OPTIONAL]: The integer data type of the field in output data files. When it is not specified, the default integer data type or the data type of the field in the

model is used. Its value is “short”, “int”, or “long”.

- unit [CHARACTER, OPTIONAL]: The unit of the field in output data files. When it is not specified, the unit of the field in the model is used.
- grid_name [CHARACTER, OPTIONAL]: The grid of the field in output data files. It should have been declared in the current configuration file. When it is not specified, the default grid or the grid of the field in the model will be used.

```
1.   <time_setting status="on" file_freq_count="2" file_freq_unit="days">
2.          <time_period status="on" freq_count="1" freq_unit="days">
3.                 <time_slots status="on" time_format="hours" slots="[1 , 2],[9,10], [17, 18]" freq_count="1"
     freq_unit="hour">
4.                        <time_points status="on" time_points="1200,2400" time_format="seconds"/>
5.                 </time_slots>
6.                 <time_slots status="on" time_format="hours" slots="[5,6],[13,15],[22,22]" freq_count="2"
     freq_unit="hour">
7.                        <time_points status="on" time_points="2400,4800" time_format="seconds"/>
8.                   </time_slots>
9.          </time_period>
10.  </time_setting>
```

Figure 13 An example of configuring output time series

3.6.1.3 XML node “time_setting”

To support various kinds of output time series in a convenient manner, C-Coupler employs the terms of period, time slot, and time point, supports flexible combination among them, and enables users to specify a periodic or an irregular output time series via the XML configuration file. Time slots can be nested in a period, so a period can contain multiple time slots. A period can also be nested in a time slot, which means that a time slot can contain a periodic time series; a period and a time slot can contain multiple time points.

A “time_period” node specifies a period (for example, L2 in Figure 13). It includes the following attributes:

- freq_unit [CHARACTER]: The unit of the period. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”, “nyear” or “years”).
- freq_count [INTEGER]: The count of the period corresponding to the period unit.

A “time_slots” node specifies a set of time slots and a period nested in the time slots (for example, L3 to L8 in Figure 13). It includes the following attributes:

- slot_unit [CHARACTER]: The time unit of the time slots. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”, “nyear” or “years”).
- slots [CHARACTER]: the values of time slots. Each time slot is a pair of integer values enclosed in “[,]”, where the first value should be no bigger than the latter one.
- freq_unit [CHARACTER]: The unit of the period nested in the time slots. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”,

"nyear" or "years").

- freq_count [INTEGER]: The count of the period corresponding to the period unit.

A "time_points" node specifies a set of time points, with the following attributes:

- point_unit [CHARACTER]: The time unit of the time points. The unit can be second ("second", "seconds", "nsecond" or "nseconds"), day ("day", "days", "nday" or "ndays"), month ("month", "months", "nmonth" or "nmonths") or year ("year", "years", "nyear" or "years").
- points [INTEGER, DIMENSION (:)]: the values of time points.

```
1.  <input_instance>
2.    <data_sources dataset_name="dataset1">
3.        <time_mapping >
4.            <period_setting status="on" start_time_in_dataset="20050101"
    start_time_format="YYYYMMDD" freq_unit="ndays" freq_count="20"   />
5.            <offset_setting status="on" offset_unit="ndays" offset_count="5"   />
6.        </time_mapping >
7.        <fields>
8.            <entry name_in_model="TSKIN2" name_in_file="GMAO-2DS__TSKIN" />
9.            <entry name_in_model="EFLUX2" name_in_file="GMAO-2DS__EFLUX" />
10.       </fields>
11.   </data_sources>
12. </input_instance>
```

Figure 14 An example of input configuration file "input_configuration_*configuration_name*.xml"

3.6.2 Configuration file for data input

Each configuration file of data input is named "input_configuration_*configuration_name*.xml" and put under the directory "CCPL_dir/datamodel/config/", where *configuration_name* corresponds to the parameter "configuration_name" of the API CCPL_register_input_handler. It is an XML file that enables users to flexibly specify input configurations (for example, Figure 14). It has a unique root node for enclosing all configuration information. The root node can have a set of child nodes "data_sources" each of which has a unique attribute as the following:

- dataset_name [CHARACTER]: the name of the time-series dataset for data input. It is also a keyword for finding the configuration file corresponding to the dataset (Section 3.6.1).

The root node contains two child nodes, e.g., an optional XML node "time_mapping" for specifying the time mapping rule between the model and the dataset (for example, L2 to L5 in Figure 14), and an XML node "fields" for specifying the fields that can be offered to the model (for example, L7 to L10 in Figure 14).

3.6.2.1 XML node "time_mapping"

The time mapping rule can be a fixed offset between the model time and data time (specified by an XML node "offset_setting"; for example, L5 in Figure 14), a fixed period for periodically using a segment of the time-series dataset (specified by an XML node "period_setting"; for example, L4 in Figure 14), or even a hybrid combination of the two.

An “period_setting” node includes the following attributes:

- freq_unit [CHARACTER]: The unit of the period. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”, “nyear” or “years”).
- freq_count [INTEGER]: The count of the period corresponding to the period unit.
- start_time_in_dataset [CHARACTER]: the start of the time segment of the dataset. Its value should match the corresponding time format.
- start_time_format [CHARACTER]: the format of the start time. Various time formats are supported, as shown in Table 17.

A “offset_seeting” node includes the following attributes:

- offset_unit [CHARACTER]: The unit of the time offset. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”, “nyear” or “years”).
- offset_count [INTEGER]: The count corresponding to the unit of time offset.

Figure 14 specifies a time mapping rule hybrid with a period of 20 days and an offset of 5 days. Such a rule determines that the first model day corresponds to the sixth day in the period, and the first and 21st model days correspond to the date 20050106 in the dataset.

```
<time_series_dataset name="dataset1">
        <data_files status="on" file_names="amip_bc.gamil.128x060.1951-2010.nc" file_type="NetCDF"/>
        <time_fields status="on" specification="file_field">
                <file_field variable="date" time_format_in_datafile="YYYYMMDD" />
                <file_field variable="datesec" time_format_in_datafile="SSSSS" />
        </time_fields>
        <horizontal_grids>
                <horizontal_grid status="on" grid_name="time_field_test_H2D_grid" specification="file_field">
                        <entry
                                file_name="amip_bc.gamil.128x060.1951-2010.nc"
                                edge_type="LON_LAT" coord_unit="degrees" cyclic_or_acyclic="cyclic"
                                dim_size1="lon" dim_size2="lat"
                                min_lon="0.0" max_lon="360.0" min_lat="-90.0" max_lat="90.0"
                                center_lon="lon" center_lat="lat"
                                annotation="time field test h2d grid"
                        />
                </horizontal_grid>
        </horizontal_grids>
        <fields>
                <field name_in_file="GMAO-2DS__TSKIN" grid_name="time_field_test_H2D_grid" />
        </fields>
</time_series_dataset>
```

Figure 15 An example of the dataset configuration file “dataset_configuration_*dataset_name*.xml”

3.6.2.2 XML node “fields”

The XML node “fields” includes a set of child nodes each of which specifies a distinct field that the model can input from the dataset. A child node has the following attributes.

- name_in_model [CHARACTER]: The field name in the model.
- name_in_file [CHARACTER]: The field name in the data files.

3.6.3 Configuration file for time-series dataset

The configuration file of a dataset for data input is named “dataset_configuration_*dataset_name*.xml” and put under the directory “CCPL_dir/datamodel/config/”, where *dataset_name* is the name of the dataset referred in the configuration files of data input (Section 3.6.2 and Figure 14). It has a unique root node “time_series_dataset” for enclosing all configuration information (for example, Figure 16). The root node has a unique attribute as the following:

- name [CHARACTER, OPTIONAL]: The name of the time series dataset.

The root node contains a set of child nodes for specifying the data files (the XML node “data_files”, e.g., L2 in Figure 15 and Figure 16), fields for time information (the XML node “time_fields”, e.g., L3 to L6 in Figure 15, and L3 in Figure 16), fields offered by the dataset (the XML node “fields”, e.g., L19 to L21 in Figure 15 and L16 to L19 in Figure 16), and the grids for fields (will be further introduced in Section 3.6.4).

```
1.  <time_series_dataset name="dataset2">
2.          <data_files status="on" files_name="GEOS5-Climate/a3_nc/evap_a3.*.nc"
    time_format_in_file_name="MMDDHH" file_type="NetCDF"/>
3.          <time_fields status="on" specification="file_name" time_point_type="start" />
4.          <horizontal_grids>
5.                  <horizontal_grid status="on" grid_name="H2D_grid1"
    specification="file_field">
6.                          <entry
7.                                  edge_type="LON_LAT" coord_unit="degrees"
    cyclic_or_acyclic="cyclic"
8.                                  dim_size1="Lon-000" dim_size2="Lat-000"
9.                                  min_lon="0.0" max_lon="360.0"
10.                                 min_lat="-90.0" max_lat="90.0"
11.                                 center_lon="LON" center_lat="LAT"
12.                                 annotation="register GC H2D grid"
13.                         />
14.                 </horizontal_grid>
15.         </horizontal_grids>
16.         <fields>
17.                 <field name_in_file="GMAO-2DS__EFLUX" grid_name="H2D_grid1" />
18.                 <field name_in_file="GMAO-2DS__TSKIN" grid_name="H2D_grid1" />
19.         </fields>
20. </time_series_dataset>
```

Figure 16 Another example of the dataset configuration file “dataset_configuration_*dataset_name*.xml”

3.6.3.1 XML node “data_files”

The unique XML node “data_files” specifies the data files in the dataset. It contains the following attributes:

- file_names [CHARACTER]: It is the common name of the data files. When there are multiple data files in the dataset, the common name should contain a unique wildcard “*”, which stands for an implicit string of time and means that the names of different data files are different only at the time string. Data files can be put under a fixed directory if the common file name includes an absolute path (e.g., the first character in the common file name is ‘/’ under Linux). Otherwise, data files should be put under the directory “CCPL_dir/datamodel/data/”.
- time_format_in_filename [STRING, OPTIONAL]: It is the time format of the time string in the names of the data files. It should be given only when there is a wildcard “*” in the common file name. Various time formats are supported, as shown in Table 17.
- file_type [CHARACTER]: It specifies the type of the data files. Currently, only NetCDF files are supported.

3.6.3.2 XML node “time_fields”

A “time_fields” node specifies how to obtain the time series of the dataset. There is only one active “time_fields” node (the value of the attribute “status” is “on”) in a configuration file. There are two ways to obtain the time series, i.e., from the time string in the names of the data files (when the attribute of “specification” is set to “file_name”; L3 in Figure 16), and from the fields of time in the data files (when the attribute “specification” is set to “file_field”; L3 to L6 in Figure 15 An example of the dataset configuration file “dataset_configuration_*dataset_name*.xml”). The attribute “time_point_type” in the “time_fields” node can be further used to specify the time information. The value of “time_point_type” can be set to “start”, “middle”, “end”, while the default value is “middle”. For example, the “start” “middle” and “end” of a day are 0th second, 43200th second and 86399th second respectively.

When the “specification” attribute is set to “file_name”, there must be a wildcard “*” in the common file name, and the “time_format_in_file_name” attribute must have been set, and each data file should have only one time point.

When the “specification” attribute is set to “file_field”, the time series can be obtained from only one field of time in the data files, or from multiple time fields combined together. Each time field is specified via a child node “file_field”, with the following attributes (for example, L4 and L5 in Figure 15 An example of the dataset configuration file “dataset_configuration_*dataset_name*.xml”):

- variable [CHARACTER]: The name of the time field in the data files.
- time_format_in_datafile [CHARACTER]: The time format corresponding to the time field. Various time formats are supported, as shown in Table 17.

3.6.3.3 XML node “fields”

There is a unique “fields” node. It consists of a set of “field” nodes, each of which corresponds to a distinct field. A “field” node contains the following attributes:

- name_in_file [CHARACTER]: The field name in the data files.
- grid_name [CHARACTER, OPTIONAL]: The grid of the field in the data files. It should have been declared in the current configuration file.

3.6.4 Configurations of grids for data input/output

As C-Coupler can handle various kinds of horizontal grid, support several kinds of vertical coordinates, and represent a 3-D grid in a “2-D + 1D” or “1D + 2-D” manner, configurations for horizontal grids, vertical coordinates, and 3-D grids were designed separately.

```
1.  <horizontal_grids>
2.      <horizontal_grid status="on" grid_name="H2D_grid1" specification="uniform_grid">
3.          <entry
4.              cyclic_or_acyclic="acyclic" coord_unit="degrees"
5.              num_lats="60" num_lons="60"
6.              min_lon="350.0" max_lon="20.0"   min_lat="-80.0" max_lat="80.0"
7.          />
8.      </horizontal_grid>
9.      <horizontal_grid status="on" grid_name=" H2D_grid2" specification="file_field">
10.         <entry
11.             file_name="GEOS5-Climate/a6_nc/a6.010100.nc"   file_type="NetCDF"
12.             edge_type="LON_LAT" coord_unit="degrees" cyclic_or_acyclic="cyclic"
13.             dim_size1="Lon-000" dim_size2="Lat-000"
14.             min_lon="0.0" max_lon="360.0" min_lat="-90.0" max_lat="90.0"
15.             center_lon="LON" center_lat="LAT"
16.         />
17.     </horizontal_grid>
18.     <horizontal_grid status="off" grid_name="H2D_grid3" specification="CCPL_grid_file">
19.         <entry
20.             file_name="GEOS5-Climate/atm_H2D_grid@atm.nc"
21.         />
22.     </horizontal_grid>
23. </horizontal_grids>
```

Figure 17 An example of specifying horizontal file grids in an XML configuration file.

3.6.4.1 Configuration of horizontal grids

A configuration file for data output or for a time-series dataset can have a unique XML node “horizontal_grids” for specifying all horizontal grids used in the configuration file. A “horizontal_grids” node includes a set of active “horizontal_grid” nodes (for example, Figure 17), each of which corresponds to a horizontal grid with a distinct name (specified via the attribute “grid_name”). There are three ways for obtaining the data of a horizontal grid, and the attribute “specification” has been designed for selecting one way.

For a regular longitude–latitude grid, the attribute “specification” can be set to “uniform_grid”, and thus the 2-D coordinate values will be calculated automatically based on the following attributes in the child node of a “horizontal_grid” node (for example, L2 to L8 in Figure

17):

- num_lons [INTEGER]: The number of different longitude (X) values in the regular grid.
- num_lats [INTEGER]: The number of different latitude (Y) values in the regular grid.
- coord_unit [CHARACTER]: The unit of coordinate values.
- cyclic_or_acyclic [CHARACTER]: It labels whether the longitude (X) direction is cyclic (with the value “cyclic”) or acyclic (with the value “acyclic”). A horizontal grid can be cyclic when the “coord_unit” is “degrees” or “radians”.
- min_lon [REAL]: the left boundary of the grid domain.
- max_lon [REAL]: the right boundary of the grid domain. The value of “min_lon” can be larger than the value “max_lon” when the “coord_unit” is “degrees” or “radians” (for example, L6 in Figure 17, where 350 degrees essentially equals -10 degrees).
- min_lat [REAL]: the bottom boundary of the grid domain.
- max_lat [REAL]: the top boundary of the grid domain.

For an irregular longitude–latitude grid, the attribute “specification” can be set to “file_field”, and thus its grid data will be read from the corresponding data file based on the following attributes in the child node of a “horizontal_grid” node (for example, L9 to L17 in Figure 17):

- file_name [CHARACTER]: The name of the file that contains the grid data. The grid data file can be put under a fixed directory if the file name includes an absolute path (e.g., the first character in the common file name is ‘/’ under Linux). Otherwise, the grid data file should be put under the directory “CCPL_dir/datamodel/data/”.
- file_type [CHARACTER]: It specifies the type of the grid data file. Currently, only NetCDF files are supported.
- coord_unit [CHARACTER]: The unit of coordinate values.
- cyclic_or_acyclic [CHARACTER]: It labels whether the longitude (X) direction is cyclic (with the value “cyclic”) or acyclic (with the value “acyclic”). A horizonal grid can be cyclic when the “coord_unit” is “degrees” or “radians”.
- edge_type [CHARACTER; IN]: The type of the edges of grid cells (“LON_LAT”， “XY”, “GREAT_ARC”, or “TriPolar”).
- min_lon [REAL]: the left boundary of the grid domain.
- max_lon [REAL]: the right boundary of the grid domain. The value of “min_lon” can be larger than the value “max_lon” when the “coord_unit” is “degrees” or “radians” (for example, L6 in Figure 17, where 350 degrees essentially equals -10 degrees).
- min_lat [REAL]: the bottom boundary of the grid domain.
- max_lat [REAL]: the top boundary of the grid domain.
- dim_size1 [INTEGER; CHARACTER]: It specifies the size of the longitude (X) dimension or the size of the horizontal grid. It can be an integer value or the name of the corresponding dimension in the grid data file.
- dim_size2 [INTEGER; CHARACTER]: If “dim_size1” has been set to the size of the horizontal grid, “dim_size2” should be set to 0. Otherwise, “dim_size2” can be set to the size of latitude (Y) dimension, or set to the name of the corresponding dimension in the grid data file.
- center_lon [CHARACTER]: It specifies the file field with the longitude (X) value of the center of each grid cell.

- center_lat [CHARACTER]: It specifies the file field with the latitude (Y) value of the center of each grid cell.
- vertex_lon [CHARACTER, OPTIONAL]: It specifies the file field with the longitude (X) value of the vertexes of each grid cell.
- vertex_lat [CHARACTER, OPTIONAL]: It specifies the file field with the latitude (Y) value of the vertexes of each grid cell. “vertex_lon” and “vertex_lat” should be set or not set at the same time.
- mask [CHARACTER, OPTIONAL]: It specifies the file field with the mask value of each grid cell.
- area [CHARACTER, OPTIONAL]: It specifies the file field with the area value of each grid cell.

```
netcdf atm_H2D_grid@atm {
dimensions:
        grid_rank = 1 ;
        grid_size = 7680 ;
variables:
        int grid_dims(grid_rank) ;
        double grid_center_lon(grid_size) ;
                grid_center_lon:long_name = "longitude" ;
                grid_center_lon:unit = "degrees" ;
        double grid_center_lat(grid_size) ;
                grid_center_lat:long_name = "latitude" ;
                grid_center_lat:unit = "degrees" ;
        int grid_imask(grid_size) ;
}
```

Figure 18 An example of C-Coupler’s default NetCDF file format for a horizontal grid.

The specification of the data of a horizontal grid can be further simplified into only a file name (“specification” is set to “CCPL_grid_file”; for example, L18 in Figure 17) when it corresponds to a NetCDF file that matches C-Coupler’s default grid data file format (for example, Figure 18).

3.6.4.2 Configuration of vertical grids

A configuration file for data output or for a time-series dataset can have a unique XML node “vertical_grids” for specifying all vertical grids used in the configuration file. A “vertical_grids” node includes a set of active “vertical_grid” nodes (for example, Figure 19), each of which corresponds to a vertical grid with a distinct name (specified via the attribute “grid_name”). There are three ways for obtaining the data of a vertical grid, and the attribute “specification” has been designed for selecting one way. Three types of vertical grids are supported, i.e., Z, SIGMA, and HYBRID. The attribute “grid_type” of a “vertical_grid” node has been designed for specifying the grid type (the attribute “grid_type” can be set to “Z”, “SIGMA” or “HYBRID”).

For a regular “Z” or “SIGMA” grid, the attribute “specification” can be set to “uniform_vertical_grid”, and thus the vertical coordinate values will be calculated automatically based on the following attributes in the child node of a “vertical_grid” node (for example, L9 to L13 in Figure 19):

- num_levels [INTEGER]: The number of levels of the vertical grid. It is also the size of the vertical grid.
- min_value [REAL]: The lower bound of the coordinate values.
- max_value [REAL]: The upper bound of the coordinate values.
- top_value [REAL, OPTIONAL]: The uniform coordinate value at the top level of the SIGMA grid.
- coord_unit [CHARACTER]: The unit of coordinate values.
- order [CHARACTER]: The order of the coordinate values automatically generated. Its value can be "ascending" or "descending".

```
1.  <vertical_grids>
2.        <vertical_grid status="on" grid_name="V1D_ grid1" grid_type="Z"
    specification="immediate_values">
3.              <entry
4.                    coord_unit="hPa"
5.                    coord_values="10, 20, 30, 50, 100, 125, 150, 175,200,250,300, 350, 400, 450, 500, 550,
    600, 650, 700, 750, 800, 850, 887, 925, 962, 1000"
6.              />
7.        </vertical_grid>
8.        <vertical_grid status="on" grid_name=" V1D_ grid2" grid_type="SIGMA"
    specification="uniform_vertical_grid">
9.              <entry
10.                   coord_unit="hPa" order="ascending"    top_value="1.0"
11.                   min_value="0.01" max_value="0.99" num_levels="20"
12.             />
13.       </vertical_grid>
14.       <vertical_grid status="on" grid_name="hybrid_grid1" grid_type="HYBRID"
    specification="file_field">
15.             <entry
16.                   file_name="GEOS5-Climate/GEOS-5_Reduced_Vertical_Grid_47_Levels.nc"
17.                   coord_unit="hPa" top_value="1.0"
18.                   coef_A="Ap" coef_B="Bp"
19.             />
20.       </vertical_grid>
21. </vertical_grids>
```

Figure 19   An example of specifying vertical grids (1-D) in an XML configuration file.

The attribute "specification" can also be set to "immediate_values", and thus the coordinate values of a vertical grid can be directly specified in the configuration file based on the following attributes in the child node of a "vertical_grid" node (for example, L3 to L6 in Figure 19):

- coord_unit [CHARACTER]: The unit of coordinate values.
- top_value [REAL, OPTIONAL]: The uniform coordinate value at the top level of a SIGMA or HYBRID grid.
- coord_values [REAL, DIMENSION (:), OPTIONAL]: The coordinate values of a Z or SIGMA grid.
- coef_A [CHARACTER, OPTIONAL]: The "coef_A" values of a HYBRID grid.
- coef_B [CHARACTER, OPTIONAL]: The "coef_B" values of a HYBRID grid.

The attribute “specification” can also be set to “file_field”, and thus its grid data will be read from the corresponding data file based on the following attributes in the child node of a “vertical_grid” node (for example, L15 to L19 in Figure 19):

- file_name [CHARACTER]: The name of the file that contains the grid data. The grid data file can be put under a fixed directory if the file name includes an absolute path (e.g., the first character in the common file name is ‘/’ under Linux). Otherwise, the grid data file should be put under the directory “CCPL_dir/datamodel/data/”.
- top_value [REAL, OPTIONAL]: The uniform coordinate value at the top level of the SIGMA grid.
- coord_unit [CHARACTER]: The unit of coordinate values.
- coord_values [REAL, DIMENSION (:), OPTIONAL]: It specifies the file field with the coordinate values of a Z or SIGMA grid.
- coef_A [CHARACTER, OPTIONAL]: It specifies the file field with the “coef_A” values of a HYBRID grid.
- coef_B [CHARACTER, OPTIONAL]: It specifies the file field with the “coef_B” values of a HYBRID grid.

```
<V3D_grids>
    <V3D_grid status="on" grid_name="V3D_lev_grid1"
        mid_point_grid_name="V3D_grid1" dimension_order="V1D+H2D">
        <horizontal_sub_grid grid_name="H2D_grid1" />
        <vertical_sub_grid grid_name="V1D_ grid2">
            <surface_field field_name_in_file="PS" />
        </vertical_sub_grid>
    </V3D_grid>
    <V3D_grid status="off" grid_name="V3D_grid2">
        <horizontal_sub_grid grid_name="handler_output_H2D_grid" />
        <vertical_sub_grid grid_name="V1D_grid1" />
    </V3D_grid>
    <V3D_grid status="off" grid_name="V3D_grid3">
        <horizontal_sub_grid grid_name="H2D_grid1" />
        <vertical_sub_grid grid_name="handler_output_V1D_grid" />
    </V3D_grid>
    <V3D_grid status="on" grid_name="V3D_lev_grid4" mid_point_grid_name="V3D_grid4">
        <horizontal_sub_grid grid_name="H2D_grid2" />
        <vertical_sub_grid grid_name="V1D_grid2">
            <surface_field field_name_in_file="GMAO-2D__PS" >
                <data_files status="on" file_names="GEOS5-Climate/i6_nc/i6.*" time_format_in_filename="MMDDHH" file_type="NetCDF"/>
                <time_fields status="on" specification="file_name" />
            </surface_field>
        </vertical_sub_grid>
    </V3D_grid>
</V3D_grids>
```

Figure 20 An example of specifying 3-D grids in an XML configuration file. The horizontal sub-grids “H2D_grid1” and “H2D_grid2” and the vertical sub grids “V1D_grid1” and “V1D_grid2” have already been specified in Figure 18 and Figure 19.

3.6.4.3 Configuration of spatial 3-D grids

A configuration file for data output or for a time-series dataset can have a unique XML node

"V3D_grids" for specifying all spatial 3-D grids used in the configuration file. A "V3D_grids" node includes a set of active "V3D_grid" nodes (for example, Figure 20), each of which corresponds to a 3-D grid with a distinct name (specified via the attribute "grid_name"). A 3-D grid consists of a horizontal sub grid and a vertical sub grid. A "V3D_grid" node can include the following attributes:

- dimension_order [CHARACTER, OPTIONAL]: It can be "V1D+H2D" that means the vertical coordinate is at the lowest dimension, and can also be "H2D+V1D" that means the vertical coordinate is at the highest dimension. When "dimension_order" is not specified in the "V3D_grid" node, its default value is "H2D+V1D".
- mid_point_grid_name [CHARACTER, OPTIONAL]: The name of the middle-point grid of the 3-D grid. When "mid_point_grid_name" is specified, the middle-point grid will be generated.

A "V3D_grid" node includes two child nodes, e.g., a node "horizontal_sub_grid" for specifying the horizontal sub grid, and a node "vertical_sub_grid" for specifying the vertical sub grid. The name of the horizontal sub grid (specified via the attribute "grid_name") can be "handler_output_H2D_grid" that means the model grid specified via the parameter "handler_output_h2d_grid_id" of the API CCPL_register_configurable_output_handler, or be the name of a horizontal grid that has been declared in the current configuration file. The name of the vertical sub grid can be "handler_output_V1D_grid" that means the model grid specified via the parameter "handler_output_v1d_grid_id" of the API CCPL_register_configurable_output_handler, or be the name of a vertical grid that has been declared in the current configuration file.

For a 3-D grid with SIGMA or HYBRID coordinate, it generally has a surface field. A "vertical_sub_grid" node can have a "surface_field" for specifying how to obtain the surface field of the 3-D grid from data files (for example, Figure 20).

3.7 Configuration file for a package of external modules

A component model can have a file named "*comp_full_name*.external_modules.xml" under the directory "CCPL_dir/config/all/external_modules" for specifying a set of packages of external modules, where "*comp_full_name*" stands for the full name of the component model. When a component model initializes an instance of an external module package via the API CCPL_external_modules_inst_init, the component model and its ancestors should have at least one configuration file with the external module package. Therefore, the configuration file corresponding to the root component model, named "overall.external_modules.xml", can be shared by all component models.

Figure 21 shows an example configuration file for packages of external modules. Each active "external_modules_package" node (for example, L3 to L10 in Figure 21) specifies the configuration of an external module package, and the unique active "fields_name_mappers" node (for example, L12 to L18 in Figure 21) enables a component model and an external module to use different names for the same argument.

- **XML node of "external_modules_package"**

An "external_modules_package" node contains the following attributes:

- name [CHARACTER]: The name of the external module package. It corresponds to the parameter "dl_name" of the API CCPL_external_modules_inst_init.
- library [CHARACTER]: The name of the default dynamic link library for the external module package. All dynamic link libraries should be put under the directory "CCPL_dir/libs/external_modules/".

An "external_modules_package" node can have a set of child nodes "procedure" each of which specifies an external module with the following attributes:

- name [CHARACTER]: The name of the current external module.
- library [CHARACTER, OPTIONAL]: The name of the dynamic link library that contains the current external module. When it is not specified, the default dynamic link library of the package will be used.
- fields_name_mapper [CHARACTER, OPTIONAL]: The name of a mapper for mapping argument names among the component model and the current external module. This mapper should have been set in the same configuration file. When no mapper is specified, the component model and the current external module should use the same name space of arguments.

■ **XML node of "fields_name_mappers"**

A "fields_name_mappers" node can have a set of child nodes each of which corresponds to a name mapper. A mapper node contains the following attributes:

- model_field_name [CHARACTER]: The name of the argument in the component model.
- procedures_field_name [CHARACTER]: The name of the argument in the external module.

```xml
<?xml version="1.0"?>
<root>
<external_modules_package name="da_individual" library="da_individual.so" status="on">
        <procedure status="on" name="da_demo_1" fields_name_mapper="mapper1"/>
        <procedure status="off" name="da_demo_2" library="new_individual.so" />
</external_modules_package>
<external_modules_package name="da_ensmean" library="da_ensmean.so" status="on">
        <procedure status="on" name="da_ensmean_demo_grid"/>
        <procedure status="on" name="da_ensmean_demo_run"/>
</external_modules_package>

<fields_name_mappers status="on">
      <fields_name_mapper   mapper_name="mapper1"   status="on">
              <entry model_field_name="sss" procedures_field_name="da1_sss" />
              <entry model_field_name="evap" procedures_field_name="gsi_evap" />
      </fields_name_mapper>
</fields_name_mappers>
</root>
```

Figure 21   An example about "*_DA_config.xml".

3.8 Configuration file for a DA algorithm instance

An DA algorithm instance should have a configuration file named "*inst*_DA_config.xml" and put under the directory "CCPL_dir/config/all/ensemble_modules", where "*inst*" means the name of the DA algorithm instance that is the parameter "inst_name" of the API CCPL_ensemble_modules_inst_init. As shown in Figure 22, such a configuration file contains a unique active XML node of "da_instance" (for example, L3 to L23 in Figure 22), which specifies the corresponding external modules (via the XML node of "external_modules"; for example, L4 in Figure 22), the ensemble of component models (via the XML node of "ensemble_components"; for example, L5-L7 in Figure 22), the periodic timer for actually running the DA algorithm instance (via the XML node of "periodic_timer"; for example, L8 in Figure 22), the statistical operations of input argument fields (via the XML node of "field_instances"; for example, L9 to L15 in Figure 22) and a rule of processing control (via the XML node of "processing_control"; for example, L16-L22 in Figure 22).

- **XML node of "external_modules"**

There should be a unique active XML node of "external_modules" for specifying the external modules corresponding to the DA algorithm. A "external_modules" node has the following attributes:

  - modules_name [CHARACTER]: The instance name of the external modules. It is the same as the parameter "inst_name" of the API CCPL_external_modules_inst_init.
  - type [CHARACTER]: The type of the external modules. It is the same as the parameter "ptype" of the API CCPL_external_modules_inst_init.
  - dll_name [CHARACTER, OPTIONAL]: The name of the dynamic link library for the external module. It is the same as the parameter "dl_name" of the API CCPL_external_modules_inst_init.

- **XML node of "ensemble_components"**

An active XML node of "ensemble_components" can be used to specify the model ensemble of running the DA algorithm. Specifically, the attribute "name" of each child node "comp_ensemble_full_name" specifies the common full name of the whole ensemble of a component model (for example, L6 in Figure 22), where "*" is the wildcard in the common name. When registering a component model to C-Coupler3, its ID is allocated and its unique full name formatted as "parent_full_name@model_ name" is generated, where "model_name" is the name of the component model, and "parent_full_name" is the full name of the parent component model (if any). The full names of different ensemble members of the same component model are similar while only different at the ensemble member number. So, the wildcard "*" in the common full name stands for the different ensemble member numbers. For an individual algorithm that will be run by each ensemble member separately, there should be no active XML node of "ensemble_components" specified.

- **XML node of "periodic_timer"**

There should be a unique active XML node of "periodic_timer" for specifying a periodic timer of actually running the DA algorithm (for example, L8 in Figure 22). A "periodic_timer" node has the following attributes:

- period_unit [CHARACTER]: The unit for specifying the period of the periodic timer. The unit must be “steps” (or “nsteps”), “seconds” (or “nseconds”), “days” (or “ndays”), “months” (or “nmonths”), or “years” (or “nyears”).
- period_count [INTEGER]: A count corresponding to the unit for specifying the period of the periodic timer.
- local_lag_count [INTEGER]: A count corresponding to the period unit, which is used to specify a lag (can be viewed as a time offset from the start time) which will impact when the periodic timer is on. For example, given a periodic timer < “period_unit”=“hours”, “period_count”=6, “local_lag_count”=3 >, its period is 6 hours, and it will not be on at the 0th, 6th, and 12th hours, but instead on at the 3rd, 9th, and 15th hours due to the “local_lag_count” of 3..

- **XML node of “field_instances”**

There should be a unique active XML node of “field_instances” for specifying the statistical operations of input argument fields (for example, L9-L15 in Figure 22). Specifically, the time statistics operations can be specified via the child node “time_processing”, and the ensemble statistics operations can be specified via the child node “ensemble_operation”. Without a “time_processing” node, instantaneous values of input argument fields will be passed to the DA algorithm. Without a “ensemble_operation” node, the DA algorithm will be used as an individual algorithm and no ensemble operation will conducted for the input argument fields.

■ **XML node of “time_processing”**

An XML node of “time_processing” is used to specify the statistical processing in each time window determined by the periodic timer (for example, L10 in Figure 22). A set of XML node of “time_processing” can be existed, but only one of them can be active at the same time.

- type [CHARACTER]: The type of statistical processing in each time window. Currently, only one type of instantaneous value (type=”inst”) is supported. If needed in the future, other types of statistical processing, such as time average, time accumulation, maximum or minimum in a time window, can be further supported.
- period_unit [CHARACTER]: The unit for specifying the period of the periodic timer. The unit must be “steps” (or “nsteps”), “seconds” (or “nseconds”), “days” (or “ndays”), “months” (or “nmonths”), or “years” (or “nyears”).
- period_count [INTEGER]: A count corresponding to the unit for specifying the period of the periodic timer.
- local_lag_count [INTEGER]: A count corresponding to the period unit, which is used to specify a lag (can be viewed as a time offset from the start time) which will impact when the periodic timer is on.

A sub XML node “field” can be used to specify the specific statistical processing type of an input model field instance with the attribute “name” and “type”.

- name [CHARACTER]: The name of field.
- type [CHARACTER]: The type of statistical processing in each time window specified for the corresponding field.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<da_instance status="on">
        <external_modules status="on" modules_name="atm_da_demo_ensgather"
type="package" dll_name="libda_ensgather_demo_atm.so"/>
        <ensemble_components status="on">
                <comp_ensemble_full_name name="ensmbe_comp_member*@atm_demo"/>
        </ensemble_components>
        <periodic_timer status="on" period_unit="seconds" period_count="7200"
local_lag_count="0"/>
        <field_instances status="on">
                <time_processing status="off" type="inst" >
                </time_processing>
                <ensemble_operation status="on" type="gather">
                        <field name="date" type="mem_1"/>
                </ensemble_operation>
        </field_instances>
        <processing_control status="on">
                <working_directory status="on" path="./da_demo"/>
                <config_scripts status="on">
                        <pre_instance_script status="on"
name="/home/sunchao/work/dafcc_demo/atm_ocn_da_demo/da_demo_run.sh"/>
                        <post_instance_script status="off" name=""/>
                </config_scripts>
        </processing_control>
</da_instance>
<da_instance status="off">
    …
</da_instance>
</root>
```

Figure 22 Another example about "*_DA_config.xml"

■ XML node of "ensemble_operation"

The "ensemble_operation" node has an attribute "type" for specifying the default type of ensemble operation for all fields, while a child node of the "ensemble_operation" node can further specify the specific type of ensemble operation for a specific argument field. The types of ensemble operation include:

- "gather": aggregating the values of an input argument from all ensemble members. An ensemble dimension will be appended to the aggregated argument. For example, given a 2-D field in each ensemble member, the corresponding aggregated field will be 2-D + 1-D, where 1-D means the ensemble dimension.
- "aver": calculating the ensemble-mean values of an input argument.

- "anom": calculating the ensemble-anomaly values of an input argument.
- "max": calculating the ensemble-maximum values of an input argument.
- "min": calculating the ensemble-minimum values of an input argument.
- "mem_*id*": using the values of an input argument from a specific ensemble member with the index number of "*id*".

- **XML node of "processing_control"**

The "processing_control" node enables users to specify a specific working directory instead of the default directory via the child node "working_directory" (for example, L17 in Figure 22). Moreover, the child node "config_scripts" enables users to specify the scripts executed before and after running the DA algorithm (for example, L18 to L21 in Figure 22).

4 Outputs of C-Coupler

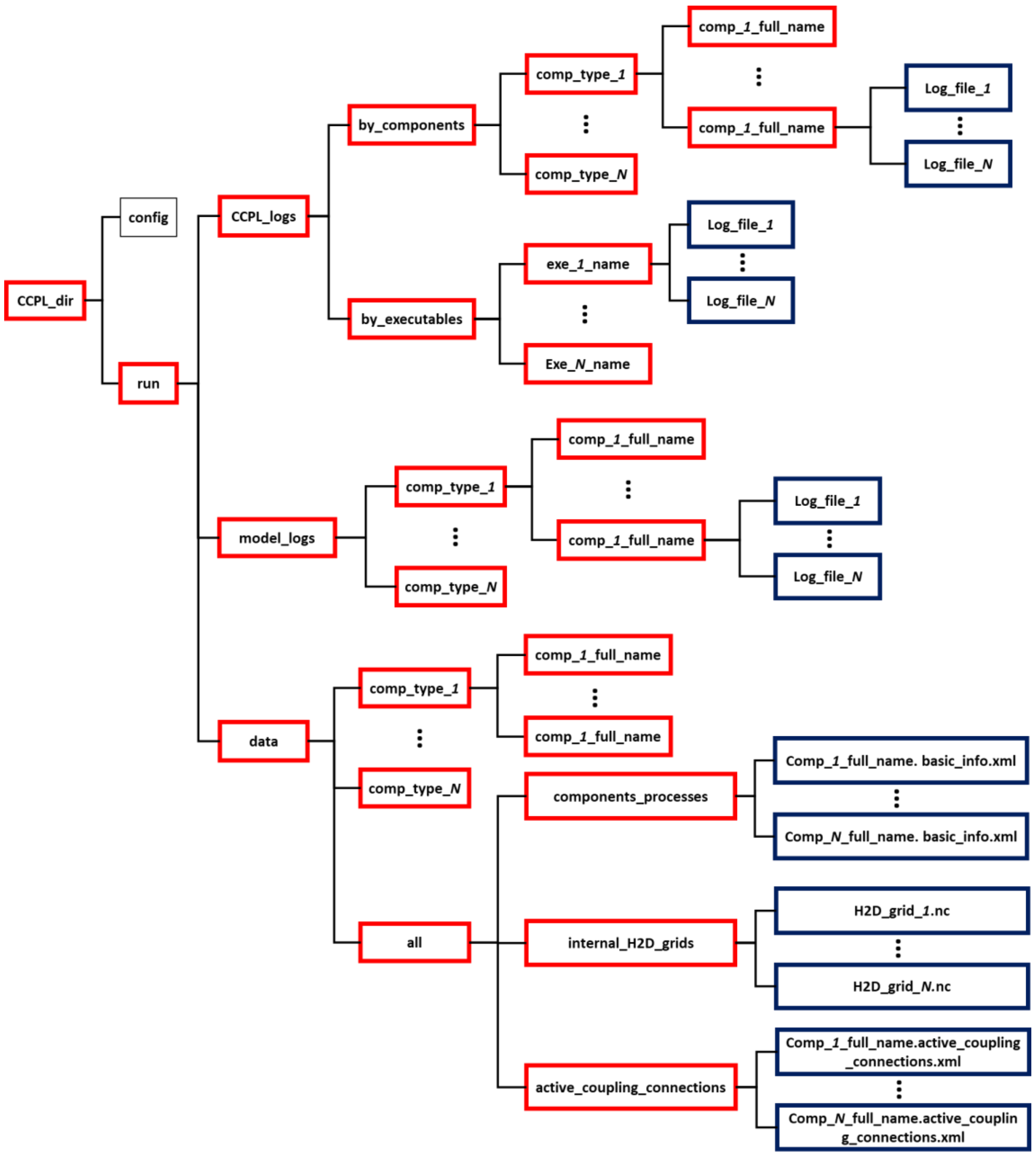


Figure 23 Directories for the outputs through C-Coupler

As shown in Figure 23, all outputs through C-Coupler are put under the directory "CCPL_dir/run", which includes three sub directories: "CCPL_logs", "model_logs" and "data". The directory "CCPL_dir/run/CCPL_logs" is for the log files output by C-Coupler or through C-Coupler APIs. It includes two sub directories: "by_components" and "by_executables". The log files under "by_components" are separated by component models, while the log files under "by_executables" are separated by executables. When C-Coupler outputs a piece of log information, the log information will be written into a corresponding log file under "by_components" and into a corresponding log file under "by_executables" separately. The log files under "by_components" enable users to check the log information of each MPI process of each component model, while the log files under "by_executables" enable users to check the latest log information of each MPI process during model integration, which can help users to effectively examine deadlocks.

The directory “CCPL_dir/run/model_logs” is for the log files directly output by component models (for example, through “print” or “write” in Fortran code). The API “CCPL_get_comp_log_file_name” (Section 2.10.4) can return the name of a log file under this directory corresponding to a MPI process of a component model, when the API “CCPL_get_comp_log_file_device” (Section 2.10.5) can further open the log file.

The directory “CCPL_dir/run/data” is for the data files output by C-Coupler. It includes a sub directory “all” and a set of sub directories each of which corresponds to a type of component models (the restart data files for a component model are put under the corresponding sub directory). Under “all”, there are three sub directories, including “component_processes” that includes the information of the MPI processes of each component model, “internal_H2D_grids” that includes the horizontal grids that have been registered to C-Coupler and “active_coupling_connections” that includes a set of XML file each of which records that active coupling connections corresponding to a component model. The active coupling connections can be further displayed in a GUI.

5 Compilation

Table 18 Environment variables in the specific Makefile under c_coupler/build

Environment variable	Description
CC	MPI C compiler
CXX	MPI C++ compiler
FC	MPI Fortran compiler
CFLAGS	The compiler flag corresponding to MPI C compiler. Please note that **"*-c*" must not be included in *CFLAGS*.**
CXXFLAGS	The compiler flag corresponding to MPI C++ compiler.
FFLAGS	The compiler flag corresponding to MPI Fortran compiler.
INCLDIR	Add directories to include file search path
SLIBS	Instruct linkers to search <dir> for libraries
CPPFLAGS	The flags for precompiling Fortran files

Table 19 Macro definitions in the C++ code of C-Coupler

Macro definition	Description
LINK_WITHOUT_UNDERLINE	When "-DLINK_WITHOUT_UNDERLINE" is specified in the C++ compiler flag (e.g., "*CXXFLAGS*" in Table 18), the underline "_" is not used in the name of each C interface in coupling_interface.cxx that is called by the Fortran API (c_coupler_interface_mod.F90). It is required to specify "-DLINK_WITHOUT_UNDERLINE" when using some compilers, such the AIX compilers.
USE_ONE_SIDED_MPI	When "-DUSE_ONE_SIDED_MPI" is specified in the C++ compiler flag (e.g., "*CXXFLAGS*" in Table 18), two-sided MPI communication (i.e., *MPI_Send/MPI_Isend*) will be disabled, while one-sided MPI communication (i.e., *MPI_put* and *MPI_get*) will be used in data transfer.
USE_PARALLEL_IO	When "-DUSE_PARALLEL_IO" is specified in the C++ compiler flag (e.g., "*CXXFLAGS*" in Table 18), parallel I/O for reading/writing fields will be used. PnetCDF should be linked when the data file format is NetCDF.

C-Coupler should be compiled into a shared library that can be further linked into various executables of models. Either an external Makefile or the specific Makefile under the directory *c_coupler/build* can be used to compile C-Coupler. For using the specific Makefile, the directory *c_coupler/build* includes an example, *build.example.sh*, which indicates a set of environment variables illustrated in Table 18. When using the specific Makefile to compile C-Coupler under the directory *c_coupler/build*, a set of objective files including the C-Coupler library named "*libc_coupler.a*" will be generated under such directory. The command "*make clean*" can easily remove the objective files under the directory *c_coupler/build*.

There are several macro definitions used in the C++ code of C-Coupler, which are further described in Table 19.