CDI-pio, CDI with parallel I/O

Deike Kleberg, Uwe Schulzweida, Thomas Jahns and Luis Kornblueh
Max-Planck-Institute for Meteorology and Deutsches Klima Rechenzentrum
Project ScalES funded by BMBF

February 9, 2017

Contents

1 Introduction **2**

2 I/O stages in the program flow **3**
- 2.1 Define **CDI** resources: `STAGE_DEFINITION` . . . 3
- 2.2 Writing in parallel: `STAGE_TIMELOOP` . . . 3
- 2.3 Cleanup **CDI** resources: `STAGE_CLEANUP` . . . 4
- 2.4 The namespace object . . . 4

3 Modes of low level writing **4**
- 3.1 `PIO_NONE`: 1 process collects and writes using `POSIX IO` . . . 5
- 3.2 `PIO_MPI`: n processes collect and write using `MPI IO` . . . 6
- 3.3 `PIO_WRITER`: $n-1$ processes collect and 1 writes using `POSIX IO` . . . 6
- 3.4 `PIO_ASYNCH`: $n-1$ processes collect and 1 writes using `POSIX AIO` . . . 7
- 3.5 `PIO_FPGUARD`: $n-1$ processes collect and write using `POSIX IO` . . . 7

4 CDI-pio modules **8**
- 4.1 Initialize parallel I/O: `pioInit` . . . 8
- 4.2 Finalize parallel I/O: `pioFinalize` . . . 9
- 4.3 Notify the end of the definition stage: `pioEndDef` . . . 9
- 4.4 Notify the end of the timestepping stage: `pioEndTimestepping` . . . 10
- 4.5 Move data to the collecting I/O server: `pioWriteTimestep` . . . 10
- 4.6 Switch between namespaces: `pioNamespaceSetActive` . . . 11
- 4.7 Offset of the local I/O subvariable: `pioInqVarDecoOff` . . . 12
- 4.8 Chunk of the local I/O subvariable: `pioInqVarDecoChunk` . . . 12

5 Internal concepts **12**
- 5.1 `MPI` communicators . . . 13

5.2 Timestepping 13

List of Figures

1.1 **CDI** `streamWriteVar` 2

3.1 Legend and pseudo code encodeNBuffer 5
3.2 `PIO_NONE` and `PIO_MPI` 6
3.3 `PIO_WRITER` 6
3.4 `PIO_FPGUARD` 7

5.1 Participation of `MPI` processes in communication "universes". 13
5.2 Pseudo code pioBuffer used in figure timestep `tsID` 13
5.3 Timestep `tsID` on model and I/O collector processes. 14

1 Introduction

The scalability of Earth System Models (ESMs) is the leading target of the ScalES project, in particular with regard to future computer development. Our work focuses on overcoming the I/O bottleneck. The Climate Data Interface (**CDI**) is a sophisticated data handling library of the Max-Planck-Institute for Meteorology with broad acceptence in the community. Its I/O is carried out synchronously relative to the model calculation. We have decided to parallelize the file writing with the **CDI**, because of the great benefit for many ESMs.

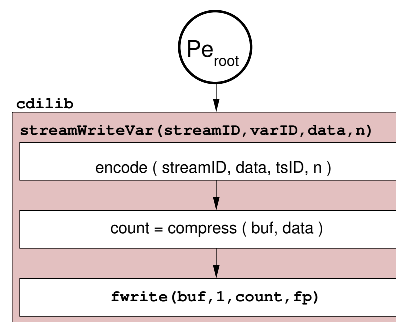


Figure 1.1: **CDI** `streamWriteVar`

We analyzed some HPC systems concerning the impact of their architecture, filesystem and MPI implementation on their I/O performance and scalability. The investigation delivered a large spectrum of results. As a consequence, we introduce different modes of low level file writing. The tasks of the I/O processes, which are now decoupled from the model calculation, are split. We destinguish into collecting the data, encode, compress and buffer it on one side and writing the data to file on the other. The extension of the **CDI** with parallel writing (**CDI-pio**) provides five different I/O modes making use of this role allocation. The modulation of I/O mode, numbers and location of processes with the best performance on a special machine can be determined in test runs.

On some systems it is impossible to write from different physical nodes to one file. Taking the architectural structures into consideration we made a key pillar for the design of **CDI-pio** from the distribution of I/O processes and files on physical nodes. If the I/O processes are located on different physical nodes, the `MPI` communicator for the group of I/O processes is split and each subgroup gets a loadbalanced subset of the files to write. On some machines this approach increases the throughput remarkably.

The application programming interface of the **CDI** is kept untouched, we introduce only a few indispensable functions and encapsulate the other developments inside the library. The models have to eliminate the special position of the root process with respect to file writing. All model processes “write” their chunk of the decomposed data and save the time former needed for gathering and low level writing.

2 I/O stages in the program flow

With the concept of I/O stages in the model program flow **CDI-pio** meets two of the main requirements for asynchronous I/O with the **CDI**: The consistency of the resources on `MPI` processes in different groups and the minimization of the communication. The program flow is divided in three stages:

`STAGE_DEFINITION`	The **CDI** resources have to be defined.
`STAGE_TIMELOOP`	Data can be moved from the model to the collecting I/O server.
`STAGE_CLEANUP`	The **CDI** resources can be cleaned up.

A listing of an example program built up of control (4.1) and model run (4.5) in chapter **CDI-pio** modules clearifies the program flow.

2.1 Define CDI resources: `STAGE_DEFINITION`

`STAGE_DEFINITION` is the default stage and starts with a call to pioInit. During this stage, the **CDI** resources have to be defined. Trying to write data with **CDI** `streamWriteVar` will lead to an error and abort the program. The stage is left by a call to pioEndDef. After leaving, any call to a **CDI** subprogramm XXX`def`YYY will lead to an error and abort the program.

2.2 Writing in parallel: `STAGE_TIMELOOP`

`STAGE_TIMELOOP` starts with a call to pioEndDef. Invocations to **CDI** `streamClose`, `streamOpenWrite`, `streamDefVlist` and `streamWriteVar` effect the local **CDI** resources but not the local file system. The calls are encoded and copied to a `MPI` window buffer. You can find a flowchart of one timestep in figure 5.3. `streamClose`, `streamOpenWrite` and `streamDefVlist` have to be called

- only for an already defined stream/vlist combination,
- in the suggested order,
- at most once for each stream during one timestep and
- before any call to `streamWriteVar` in that timestep.

All four **CDI** stream calls require that the model root process participates. Disregards to this rules will lead to an error and abort the program. The implication also holds for attempts to define, change or delete **CDI** resources during `STAGE_TIMELOOP`. Therefor it is necessary to switch stages before cleaning up the resources. A call to pioEndTimestepping closes `STAGE_TIMELOOP`.

2.3 Cleanup CDI resources: STAGE_CLEANUP

STAGE_CLEANUP is launched by invoking pioEndTimestepping. In this stage, the **CDI** resources can be cleaned up. Trying to write data with **CDI** streamWriteVar will now lead to an error and abort the program.

2.4 The namespace object

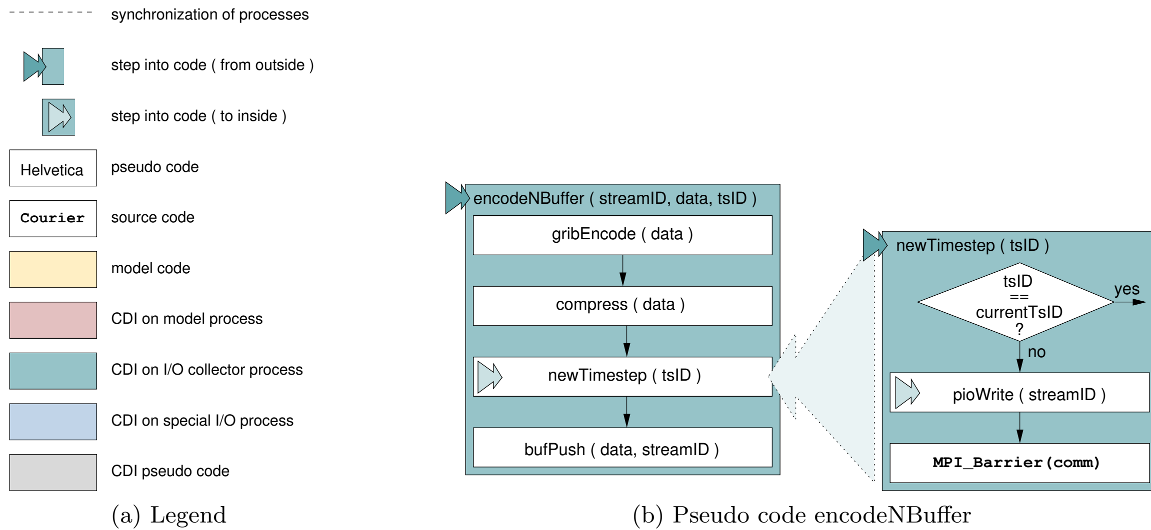
For some models the concept of stages is to narrow. In order to meet this requirement we introduce namespaces. A namespace

- has an identifier,
- is mapped to a **CDI** resource array,
- indicates, if the model processes write locally or remote,
- has an I/O stage and
- is the active namespace or not.

A call to pioInit initializes the namespace objects with two of the arguments given by the model processes, the number of namespaces to be used and an array indicating if they obtain local or remote I/O. Invoking pioNamespaceSetActive switches the namespace so that subsequent **CDI** calls operate on the resource array mapped to the chosen namespace. The namespaces are destroyed by pioFinalize. If the model uses the **CDI** serially, exactly one namespace supporting local writing is a matter of course. To save overhead, it is preferable to work with one namespace.

3 Modes of low level writing

The I/O performance and scalability of a supercomputer depends on the combination of the hardware architecture, the filesystem and the MPI implementation. We made testruns on several machines invoking MPI_File_iwrite_shared, the obvious way of parallel file writing. Especially on the IBM Blizzard, naturally in our primary focus, the benchmark programs achieved surprisingly poor results. Accordingly the **CDI** has to provide possibilities to write files in parallel using POSIX IO. The tasks formerly carried out by the root process are split into the subtask gather, encode and compress on one side and write the files on the other. The variable partition of these subtasks in the group of I/O server led us to five modes of low level writing. **CDI-pio** is backwards compatible due to the differentiated behavior of the **CDI** calls streamClose, streamOpenWrite, streamDefVlist and streamWriteVar on model side, depending on local writing and I/O stage. The stream functions were primarily written for low level file writing, as a matter of course also the collecting I/O processes invoke them. This makes the I/O modes to another key to the program flow of the **CDI** stream calls.



(a) Legend

(b) Pseudo code encodeNBuffer

Figure 3.1: Legend and pseudo code encodeNBuffer

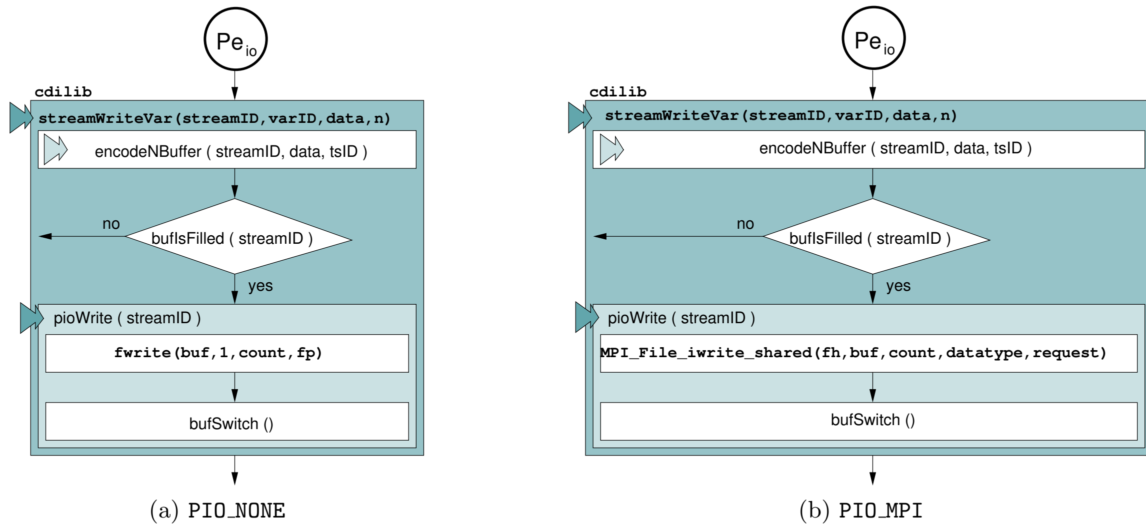
On the I/O processes the excecution of the subprogram `streamWriteVar` is controlled by the I/O modes. To clearify the functioning we use pseudo code and flowcharts as you can see in figure 3.1. The command `encodeNBuffer` abstracts the encoding, compressing and buffering of the data for a variable. The data in a `GRIB` file may not be mixed by time, so we need a command `newTimestep` to manage the flushing of the output buffers on the I/O server side. To achieve this, a `MPI_Barrier` is used.

I/O modes provided by **CDI-pio**:

- `PIO_NONE` one process collects, transposes, encodes, compresses, buffers and writes using `C fwrite`.
- `PIO_MPI` all processes collect, transpose, encode, compress, buffer and write using `MPI_File_iwrite_shared`.
- `PIO_ASYNCH` one process writes the files using low level `POSIX_AIO`, the others collect, transpose, encode, compress and buffer.
- `PIO_FPGUARD` one process guards the fileoffsets, all others collect, transpose, encode, compress and write using `C fwrite`.
- `PIO_WRITER` one process writes the files using `C fwrite`, the others collect, transpose, encode, compress and buffer.

3.1 `PIO_NONE`: 1 process collects and writes using `POSIX IO`

The I/O mode `PIO_NONE` can only be run with one I/O process per physical node. This process collects, encodes, compresses, buffers and writes the data to the files attributed to his node. For low level file writing the `C` standard `fwrite` is used. The advantages in comparison with the former serial writing are that the writing is done asynchronous with respect to the calculation and that the data is buffered. In addition it can be executed in parallel spread over physical nodes.



(a) PIO_NONE (b) PIO_MPI

Figure 3.2: PIO_NONE and PIO_MPI

3.2 PIO_MPI: n processes collect and write using MPI IO

Data access using MPI is the straight forward way to parallel file manipulation. With MPI_File_iwrite_shared the processes have a shared file pointer available. The function is non-blocking and split collective. Like PIO_NONE the I/O mode PIO_MPI has no division of task within the I/O group, all processes collect, encode, compress, buffer and write to file. Writing in this I/O mode strongly depends on the MPI implementation, the buffers used internally are of major importance for the performance of writing.

3.3 PIO_WRITER: $n - 1$ processes collect and 1 writes using POSIX IO

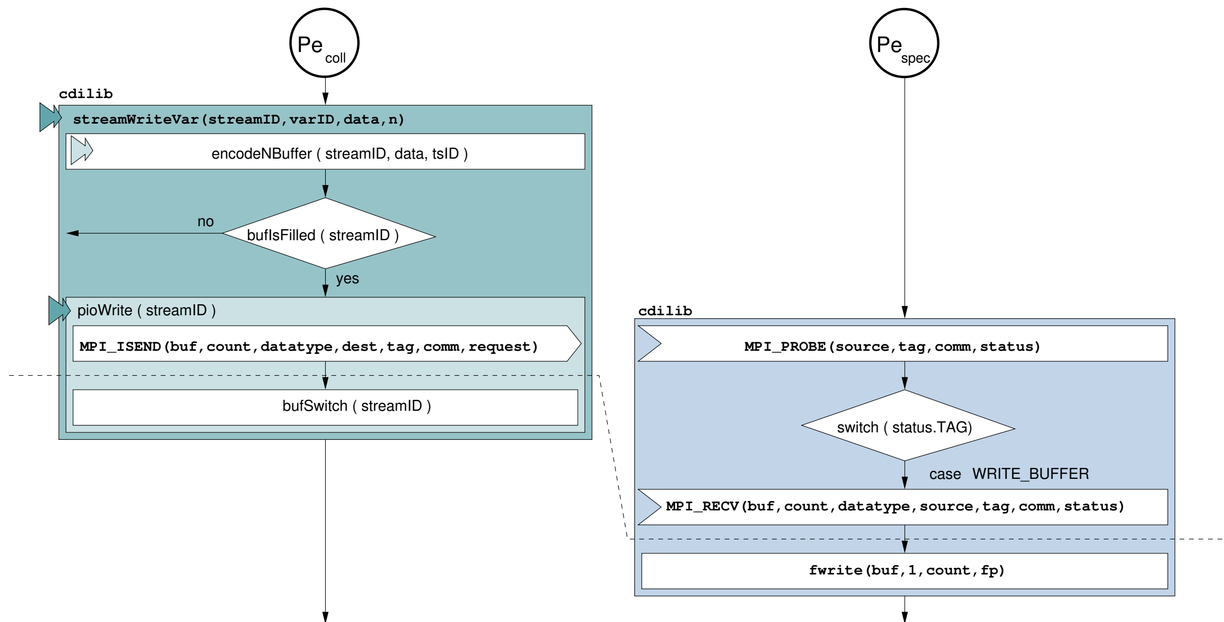


Figure 3.3: PIO_WRITER

If the I/O mode PIO_WRITER is chosen, the subtasks of writing are split between the I/O processes. Just one process per physical node does the low level writing while the others collect, encode, compress and buffer the data. The writer is the process with the highest rank within the I/O

group on one physical node. Originating from pioInit he invokes a backend server function, which he does not leave until he received messages from all collecting I/O processes to finalize. A collector gets data from the calculating model processes via MPI RMA communication, and, after encoding and compressing it, pushes it to a double buffer. If the buffer is filled, the contained data is send via MPI_Isend to the writer, the collector switches to the other buffer and continues his job. Before sending the data he has to wait for a potentially outstanding MPI_Request. This might happen if the writer or the buffers used by MPI are overcommited and indicates that the ratio of collectors and writers has to be checked. The writer is polling using MPI_Probe to look for incoming messages from the collectors. One message per collecting process is tagged with the finalize command. All other messages contain a stream identifier, a buffer with data to be written and a file manipulation instruction. There are three kinds of this commands: 1. open a file and write the data to it, 2. write the data to an open file and 3. write the data to an open file and close it afterwards. For the file writing C standard fwrite is used.

3.4 PIO_ASYNCH: $n-1$ processes collect and 1 writes using POSIX AIO

The I/O mode PIO_ASYNCH is similar to PIO_WRITER, it only differs in the method used for low level file writing. The asynchronous nonblocking I/O can be overlapped with processing, write orders are passed to the operating system.

3.5 PIO_FPGUARD: $n-1$ processes collect and write using POSIX IO

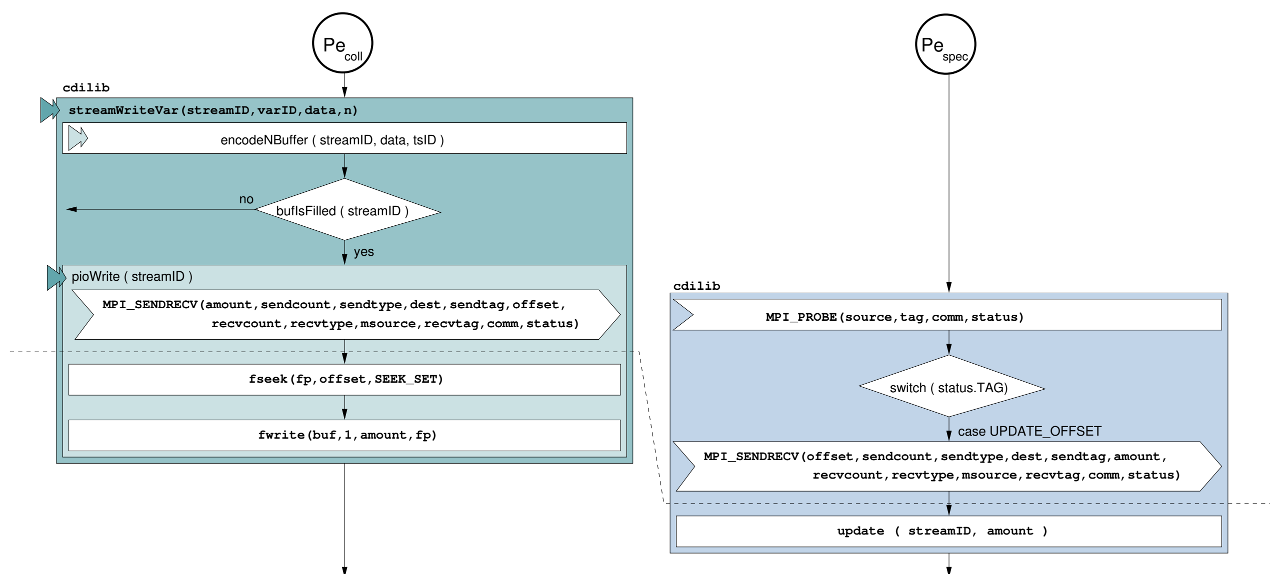


Figure 3.4: PIO_FPGUARD

Writing a huge amount of data with a fixed file offset is a very fast way of file writing. In this I/O mode one I/O process per physical node is spent to administrate the file offsets while the others do all the subtasks former defined for the parallel I/O. The functionality of this collaboration is similar to PIO_WRITER. Originating from pioInit the process with the highest rank calls a backend server function in which it is busy waiting for messages. The collecting I/O processes get data from the calculating model processes via MPI RMA communication. The data is encoded, compressed and buffered. If the buffer is filled, the collector sends the count of the contained data to the “file pointer guard” and gets a file offset back. With the received offset the collector writes the data to file using C standard fwrite and goes on with his job. One message per collecting process is tagged with the finalize command. All other messages needed for the communication

between the "file pointer guard" and the collectors contain a stream identifier, a numeric value holding the amount of buffered data respectively the file offset and a command. There are three kinds of commands: 1. offset wanted for a file that will be newly opened, 2. offset wanted for an open file and 3. offset wanted for a file that will be closed after writing.

4 CDI-pio modules

4.1 Initialize parallel I/O: `pioInit`

The function `pioInit` initializes the parallel I/O with **CDI**, it launches the `STAGE_DEFINITION`. `pioInit` defines a control object for the `MPI` communicators (see figure 5.1) and triggers their initialization. After starting the I/O server, `pioInit` receives a message from each I/O process, containing information about its location on a physical node and its function as a collector of data or a backend server. The first information is stored in the control object, the latter is used to construct the communicators for the data transfer. Furthermore, `pioInit` defines and initializes an control object for the namespaces. The call `pioInit` is collective for all `MPI` processes using the **CDI**. If the model employs the **CDI** serially, a call to `pioInit` has no effect.

Usage

```
INTEGER FUNCTION pioInit ( INTEGER commGlob, INTEGER nProcsIO,
                           INTEGER IOMode, INTEGER nNamespaces,
                           INTEGER hasLocalFile ( nNamespaces ));
```

`IN commGlob`	`MPI` communicator (handle).
`IN nProcsIO`	The number of `MPI` processes that shall be used for I/O.
`IN IOMode`	The mode for the I/O. Valid I/O modes are `PIO_NONE`, `PIO_MPI`, `PIO_WRITER`, `PIO_ASYNCH` and `PIO_FPGUARD`.
`IN nNamespaces`	The number of used namespaces on the model side.
`IN hasLocalFile`	A logical array with size `nNamespaces` indicating whether the model processes write locally or let the I/O server write.

Result

Upon successfull completion `pioInit` returns a `FORTRAN` handle to a `MPI` communicator including only the calculating model processes.

Errors

If an error occurs, `pioInit` cleans up, finalizes `MPI` and exits the whole program.

The arguments of `pioInit` subject to some constraints.

commGlob	has to be a valid handle to a MPI communicator whose group includes all processes that will work on the **CDI** resources.
nProcsIO	== 1 per physical node, if IOMode == PIO_NONE, <= sizeGlob /2 otherwise, with sizeGlob = number of processes in commGlob, >= 2 per physical node, if IOMode ∈ { PIO_WRITER, PIO_ASYNCH, PIO_FPGUARD}.

Example

Here is an example using pioInit to start parallel I/O in PIO_NONE mode.

```
INCLUDE 'cdi.inc'
INCLUDE 'mpif.h'
   ...
INTEGER commModel, error
   ...
CALL MPI_INIT ( error )
   ...
! Initialize asynchronous I/O with CDI
! Definition stage for CDI resources
commModel = pioInit ( MPI_COMM_WORLD, 1, PIO_NONE, 1, (/ 0 /))
   ...
CALL MODELRUN ( commModel )
   ...
! End cleanup stage for CDI resources
! Finalize asynchronous I/O with CDI
CALL pioFinalize ()
   ...
CALL MPI_FINALIZE ( error )
```

4.2 Finalize parallel I/O: pioFinalize

The function pioFinalize finalizes the parallel I/O. It cleans up the namespaces and sends a message to the collector processes to close down the I/O server. The buffers and windows which where needed for MPI RMA are deallocated. At last pioFinalize frees the MPI communicators (see figure 5.1) and destroys the control object. The call pioFinalize is collective for all model processes having invoked pioInit. If the model employs the **CDI** serially, a call to pioFinalize has no effect.

Usage

```
SUBROUTINE pioFinalize ();
```

4.3 Notify the end of the definition stage: pioEndDef

The end of the definition stage for the **CDI** resources in a namespace is marked with a call to pioEndDef. pioEndDef changes the state of the active namespace to STAGE_TIMELOOP. During this stage, a new definition of a **CDI** object as well as deletion or changing of members of known objects in the active namespace will lead to an error and shut the program down. There is one exception: To write files for a defined variable list individually for disjunct time intervalls the three calls

- SUBROUTINE streamClose (streamID_1),

- `INTEGER FUNCTION streamOpenWrite ( filename, filetype )` and
- `SUBROUTINE streamDefVlist ( streamID_2, vlistID_1 )`

can be used once at the beginning of a timestep, one time for each stream. When used with remote writing in this stage, these subprogram effect the local **CDI** resources but not the local file system. The calls are encoded and buffered in the MPI window buffer to be fetched by the collector processes. You can find a flowchart of a timestep on model and collecting I/O processes in figure 5.3.

`pioEndDef` balances the load of the variable data among the data collecting I/O server and stores a mapping in the variables **CDI** resource. Among the model processes `pioEndDef` decomposes the variable in rank order, laid-out in linear memory as needed for the file output. An index array with offset and chunk is also written to the variables resource. With this result and the information about the decomposition for the model calculation it defines I/O transposition templates that will be used while writing the data. `pioEndDef` copies the **CDI** object array to a buffer and sends it to the collecting I/O server for them to possess the same resource handles.

`pioEndDef` calculates the memory requirement for the MPI windows and buffers needed for RMA out of the dimensions stored in the **CDI** resources. As a last step it allocates the buffers and creates the MPI windows. If the model uses the **CDI** serially, a call to `pioEndDef` has no effect.

Usage

```
SUBROUTINE pioEndDef ();
```

4.4 Notify the end of the timestepping stage: `pioEndTimestepping`

With the function `pioEndTimestepping` the end of the time integration is notified. `pioEndTimestepping` sets the state of the active namespace to `STAGE_CLEANUP`. In this stage it is possible to clean up the **CDI** resources. There is no transfer of data or subprogram calls anymore, an attempt will lead to an error and shuts the whole program down. If the model uses the **CDI** serially, a call to `pioEndTimestepping` has no effect.

Usage

```
SUBROUTINE pioEndTimestepping ();
```

4.5 Move data to the collecting I/O server: `pioWriteTimestep`

The subroutine `pioWriteTimestep` exposes the MPI windows to the data collecting I/O server. They start to move the data from the MPI window buffers of the calculating model processes to their own memory while the model processes can go on doing their job. The buffers contain the encoded subprogram calls and the variable data written by calls to `streamClose`, `streamOpenWrite`, `streamDefVlist` and `streamWriteVar`. If the model uses the **CDI** serially, a call to `pioWriteTimestep` has no effect.

Usage

```
SUBROUTINE pioWriteTimestep ( INTEGER tsID, INTEGER vdate, INTEGER vtime );
```

`IN tsID`	Timestep identifier
`IN vdate`	Verification date (YYYYMMDD)
`IN vtime`	Verification time (hhmmss)

Example

```
INCLUDE 'cdi.inc'
INCLUDE 'mpif.h'
    ...
SUBROUTINE MODELRUN ( commModel )

  INTEGER streamID, vlistID, varID, tfID, ntf, tsID, nts, vdate, vtime

  ! Definition stage for CDI resources
  streamID = streamOpenWrite ( filename, filetype )
    ...
  CALL streamDefVlist(streamID, vlistID)

  ! End definition stage for CDI resources,
  CALL pioEndDef ()

  ! Timestepping stage
  DO tfID = 0, ntf-1
    IF ( tfID ) THEN
      CALL streamClose ( streamID )
      streamID = streamOpenWrite ( filename[tfID], filetype )
      CALL streamDefVlist ( streamID, vlistID )
    ENDIF

    DO tsID = 0, nts-1
      ...
      CALL streamWriteVar(streamID, varID, varData, nmiss)

      ! Expose encoded and buffered subroutine calls and data
      ! to remote memory access by collecting I/O server.
      CALL pioWriteTimestep ( tsID, vdate, vtime )
    END DO
  END DO

  ! End timestepping stage
  CALL pioEndTimestepping ()

  ! Cleanup stage
  CALL streamClose(streamID)
    ...
  CALL vlistDestroy(vlistID)

END SUBROUTINE MODELRUN
```

4.6 Switch between namespaces: `pioNamespaceSetActive`

The subroutine `pioNamespaceSetActive` sets the active namespace to the argument `INTEGER IN nspID`. The namespace objects are defined with the call to `pioInit`. A namespace object has the entries

`INTEGER nspID`	Namespace identifier,
`INTEGER hasLocalFile`	indicating whether the namespace supports local writing, ∈ {`TRUE`, `FALSE`}, == `TRUE` by default,
`INTEGER stageCode`	is set by calls to the subprograms `pioEndDef` and `pioEndTimestepping`, ∈ {`STAGE_DEFINITION`,`STAGE_TIMELOOP`,`STAGE_CLEANUP`}, == `STAGE_DEFINITION` by default.

If the model uses the **CDI** serially, a call to `pioNamespaceSetActive` has no effect.

Usage

```
SUBROUTINE pioNamespaceSetActive ( INTEGER nspID )
```

`IN nspID`	Namespace identifier

4.7 Offset of the local I/O subvariable: `pioInqVarDecoOff`

Obsolete.

Usage

```
INTEGER FUNCTION pioInqVarDecoOff ( INTEGER vlistID, INTEGER varID )
```

`IN vlistID`	Variable list identifier
`IN varID`	Variable identifier

4.8 Chunk of the local I/O subvariable: `pioInqVarDecoChunk`

Obsolete.

Usage

```
INTEGER FUNCTION pioInqVarDecoChunk ( INTEGER vlistID, INTEGER varID )
```

`IN vlistID`	Variable list identifier
`IN varID`	Variable identifier

5 Internal concepts

5.1 MPI communicators

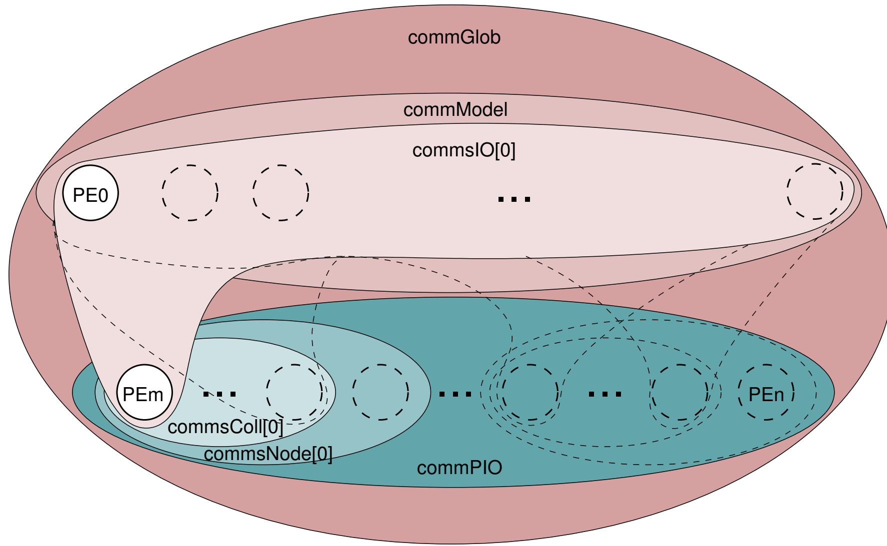


Figure 5.1: Participation of MPI processes in communication “universes”.

5.2 Timestepping

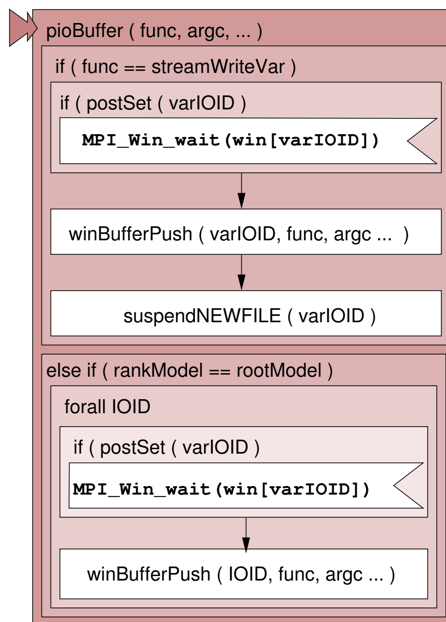


Figure 5.2: Pseudo code pioBuffer used in figure timestep `tsID`

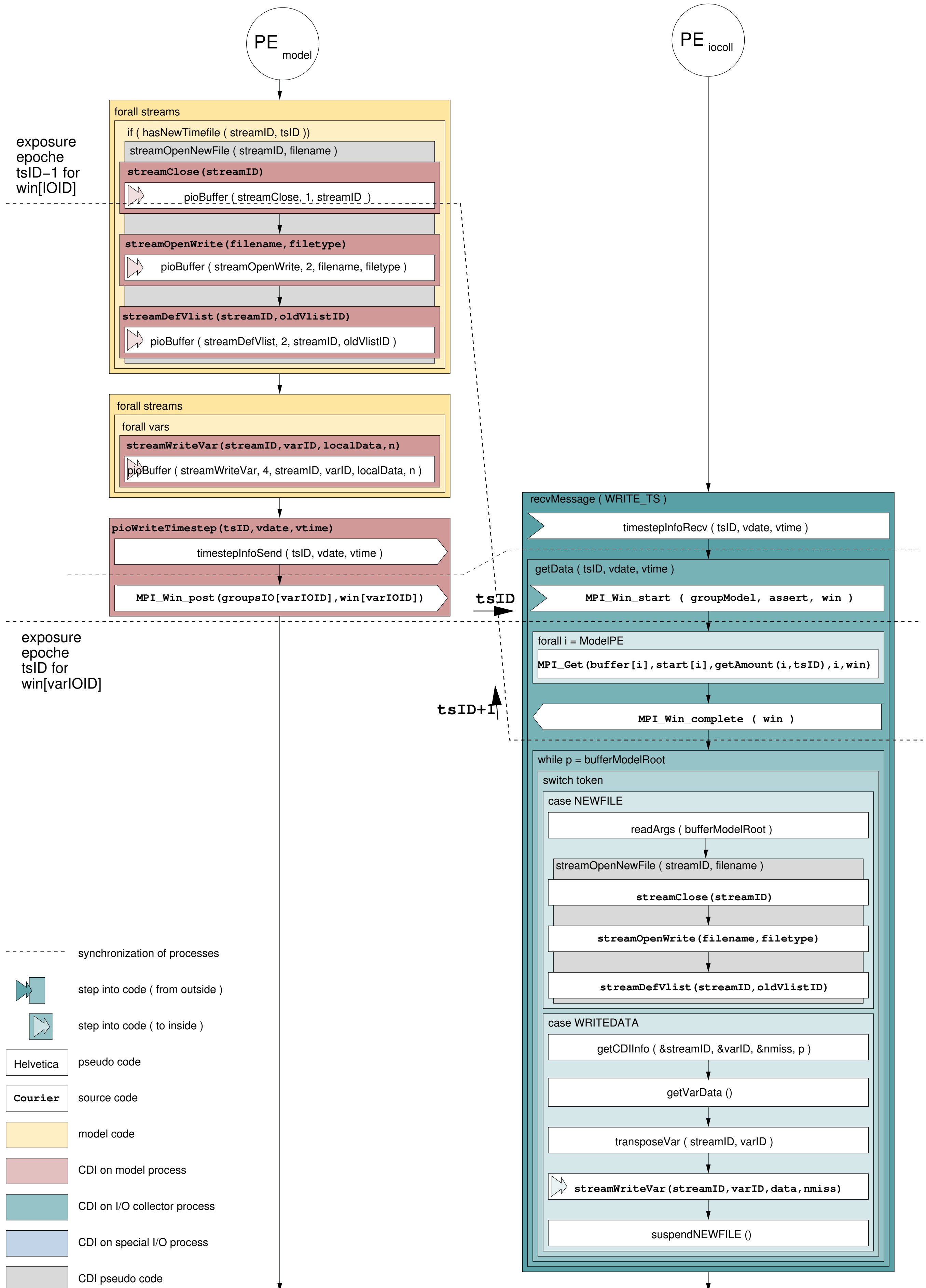


Figure 5.3: Timestep `tsID` on model and I/O collector processes.

Index

E

encodeNBuffer 5

L

legend 5

N

namespace 4

P

PIO_ASYNCH 7
pioEndDef 9
pioEndTimestepping 10
pioFinalize 9
PIO_FPGUARD 7
pioInit 8
pioInqVarDecoChunk 12
pioInqVarDecoOff 12
PIO_MPI 6
pioNamespaceSetActive 11
PIO_NONE 5
PIO_WRITER 6

S

STAGE_CLEANUP 4
STAGE_DEFINITION 3
STAGE_TIMELOOP 3

T

timestep 14