ARPEGE/ALADIN/AROME IO in 39t1

Table of contents

1 Introduction

This document describes the IO subsystem of ARPEGE/ALADIN/AROME for cycle 39t1. Only historic data IO is covered here, with a few words on Fullpos IO.

2 Organization of the code

2.1 Field descriptors

`IOFLDDESC_MOD` contains the definition of the structures designed to contain the meta-data describing fields to be read or written :

- `IOFLDDESC` contains all information about the field :
  - prefix, level, suffix
  - spectral/grid-point
  - number of bits to be used for compression
  - whether the field contains undefined values
  - ...
- `IOCPTDESC` contains additional information about a compressed field :
  - name of the article to be written to LFI
  - length of the compressed data
  - MIN/MAX/AVG of the field

2.2 IO buffers

`IOMULTIBUF_MOD` contains the definition of a structure used to store field data. Its name is `IOMULTIBUF`.

In order to avoid doing unnecessary copies, it is necessary not to use a big two dimensional array, but rather a list of two dimensional arrays. In Fortran, this is achieved by defining arrays of `IOMULTIBUF` objects.

`IOMULTIBUF` arrays have some sort of accessor methods :

`IOMULTIBUF_SIZE_IDX`	Compute the total number of fields in a `IOMULTIBUF` array
`IOMULTIBUF_INIT_IDX`	Initialize an iterator on a `IOMULTIBUF` array
`IOMULTIBUF_INCR_IDX`	Increase the iterator value
`IOMULTIBUF_COMP_IDX`	Compute the direct addressing values indexed by field

2.3 Extracting/loading fields

Model field copy into buffers is implemented in the following modules :

Module	Description
`IOGRIDUA_MOD`	Upper air grid-point fields
`IOFU_MOD`	Cumulated fluxes
`IOGRIDVA_MOD`	Climatological fields
`IOXFU_MOD`	Instantaneous fluxes
`IOGRCLIA_MOD`	Climatological fields
`IOGRIDA_MOD`	Surface physical fields
`IOSPECA_MOD`	Spectral fields

Each of these modules has the very same layout :

- a routine for counting the fields to be copied into buffers (e.g. `IOGRIDUA_COUNT`)
- a routine for retrieving the meta-data related to the fields to be extracted (e.g. `IOGRIDUA_SELECTD`)
- a routine for copying data from/to model variables to/from buffers (e. g. `IOGRIDUA_SELECTF`)

`IOSPECA_MOD` has additional routines, because of the transformations we need to apply on the data before doing spectral IO (in ARPEGE, vorticity and divergence are written, while in AROME/ALADIN, U and V are written).

2.4 MFIOOPTS

This structure is defined and initialized in `mfioopts_mod.F90`. It is supposed to contain all parameters related to how to perform output of historic files, except parameters describing the IO server configuration (described in another section).

2.5 Code cleaning

The reorganization of the code made it possible to clean the following routines :

Set-up	Output
`SUGRIDA`	`WRGRIDA`
`SUGRIDUA`	`WRGRIDUA`
`SUGRIDVA`	-

SUGRCLIA	-
SUSPECA	WRSPECA
-	WRXFU
-	WRFU

Note that routines such as **SUGRIDA** and **WRGRIDA** now use the same building blocks; they both rely on **IOGRIDA** to count the fields, retrieve their meta-data and extract field data to buffers.

All routines dealing with grid-point data rely on **WRGP2FA** for writing their fields, and on **RDFA2GP** for reading.

New routines have also been implemented using this new structure for handling SURFEX historic data :

Set-up	Output
SUGRIDSFX	WRSFX

SURFEX IO will be described in detail later in this document.

3 Grid-point/spectral

Spectral data can be read/written in grid-point format; this is activated using the following flags:

```
&NAMCT0
  LWRSPECA_GP       ! write spectral fields in grid-point format
  LSUSPECA_GP       ! read spectral fields in grid-point format
  LWRSPECA_GP_UV    ! write U/V instead of VOR/DIV
  LSUSPECA_GP_UV    ! read U/V instead of VOR/DIV
/
```

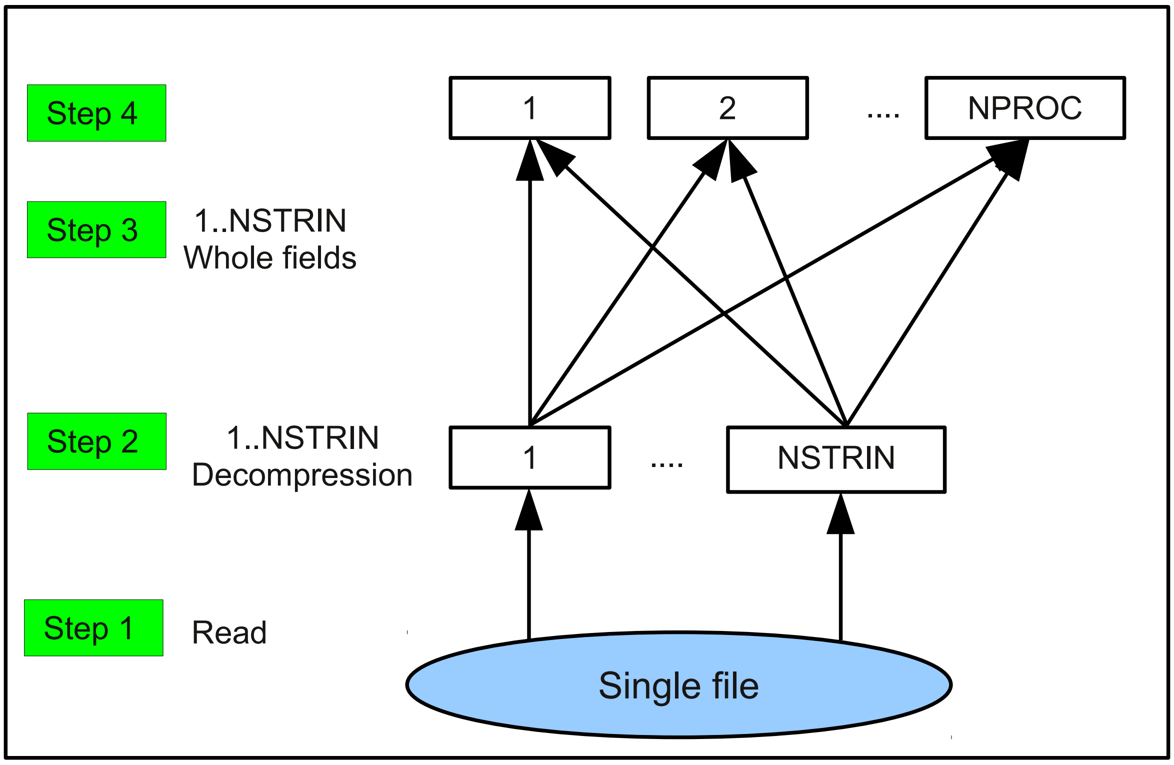
When **LWRSPECA_GP** (resp. **LSUSPECA_GP**) is true, **WRSPECA_GP** (resp. **SUSPECA_GP**) is called in place of **WRSPECA** (resp. **SUSPECA**). **WRSPECA_GP** and **SUSPECA_GP** are implemented using basic routines from **IOSPECA** and the spectral transforms.

Note that **LSUSPECA_GP** and **LSUSPECA_GP_UV** could be set up automatically by probing the input file for U/V field presence and representation in spectral coefficients (using **FANION**), but this is not the case for now.

4 Traditional input/output

4.1 Grid-point input

The `RDFA2GP` routine reads input fields from FA files in parallel (on `NSTRIN` MPI tasks) and distributes fields on all model tasks. It is able to read data from several input files.

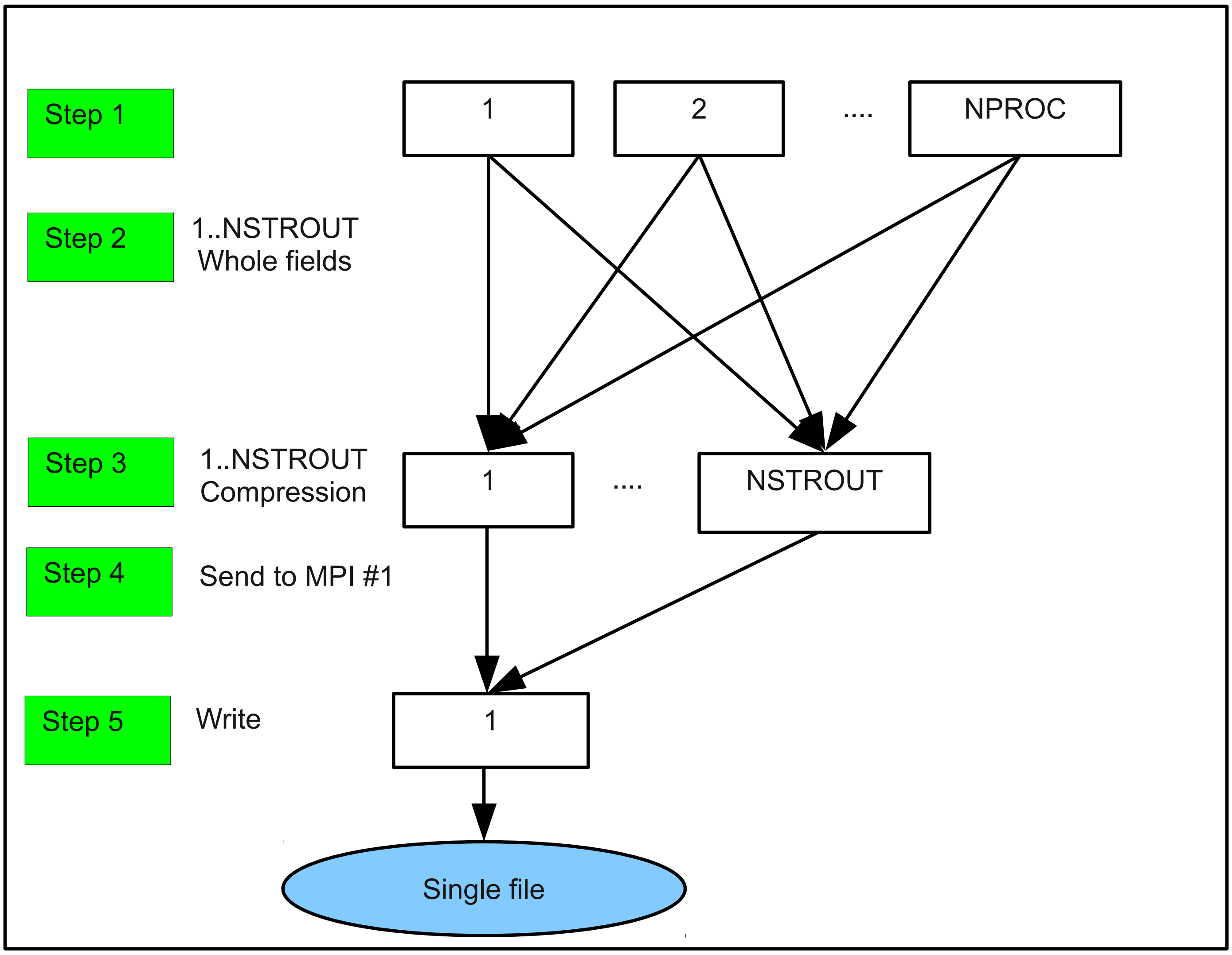


4.2 Grid-point output

What we call traditional output is handled directly by model tasks. It consist of the following steps :

- re-create whole fields on `NSTROUT` processors
- compress whole fields on `NSTROUT` processors
- send compressed fields to MPI #1
- MPI #1 writes to a single file

This is illustrated on the following diagram.



Traditional output is implemented in **`WRSPECA`**, **`WRGP2FA`** (model historic data), **`WRSFP`**, **`WRGP2FAFP`** (Fullpos data). **`WRSPECA`**, **`WRGP2FA`** and **`WRGP2FAFP`** rely on **`WRGAFLNM`** for the last two steps (4 and 5).

4.3 Grid-point output alternatives

Step 2 for grid-point fields can be performed using different MPI communications patterns :

- nominal mode : **MPI_SEND**/**MPI_RECV**, possibly with OpenMP (controlled by **LWRGRIDOPENMP**).
- using **MPI_GATHERV**, with OpenMP (controlled by **L_GATHERV_WRGP** and **LWRGRIDOPENMP**).
- using **MPI_ALLTOALLV**, controlled by **LWRGRIDALLTOALL**

There are several routines doing output of grid-point fields : **WRGRIDA**, **WRGRIDUA**, **WRXFU**, **WRFU**. All of them call **WRGP2FA**. The aforementioned routines are specialized : each of them handle a particular kind of grid-point field (for instance **WRGRIDUA** takes care of upper-air fields).

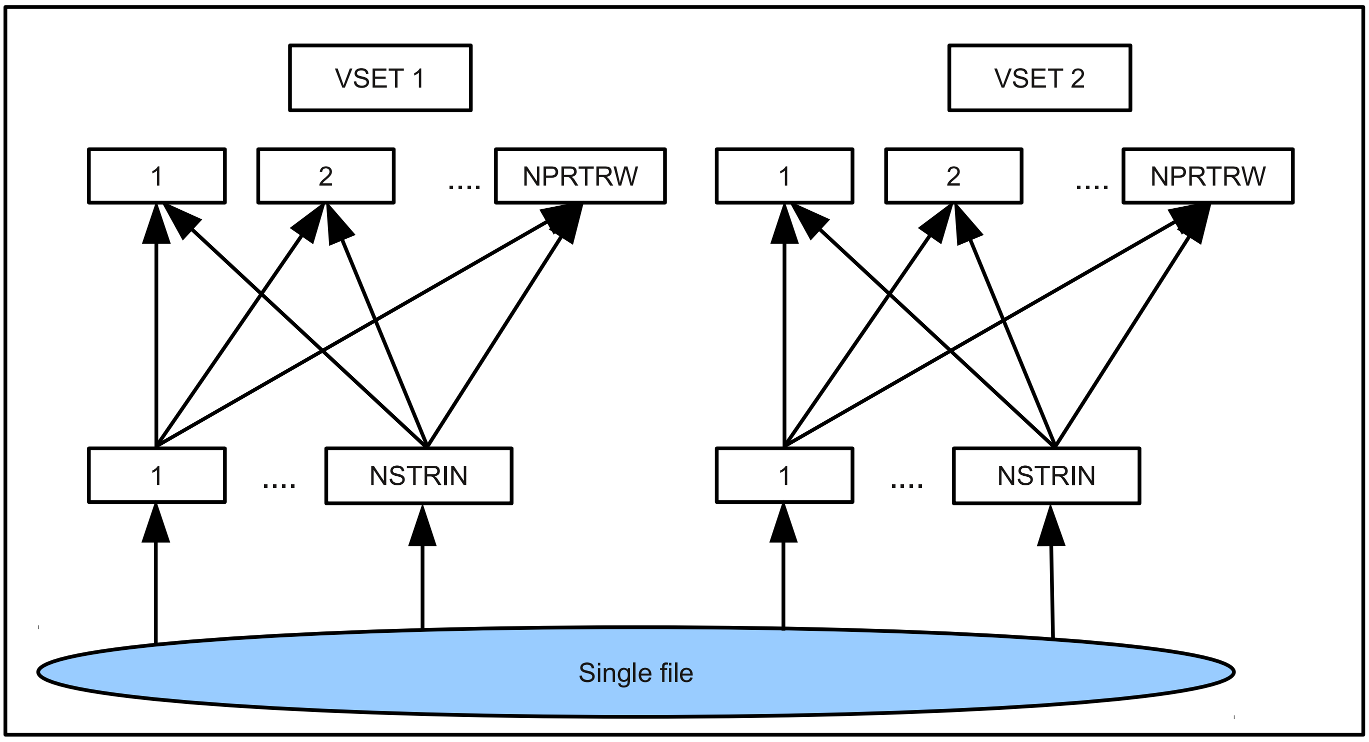
It is possible to write all grid-point fields (but SURFEX's) in one go and get more parallelism of field compression. **WRGRIDALL** does this job, under the control of **LUSEWRGRIDALL**.

In principle, it should be possible to write a **SUGRIDALL**, whose purpose would be to read all grid-point fields at once.

4.4 Spectral fields input/output

Reading or writing spectral fields is different from reading/writing grid-point fields, because of the distribution of spectral data. When a grid-point field is distributed, each MPI task gets a chunk of the field. This is not the case with spectral distribution: fields are distributed by level and wave-number.

It is then possible to have spectral V-sets working in parallel, as illustrated on the following diagram (only two V-sets, MPI tasks indexed by their W-set number). This is what is done in **WRSPECA** and **SUSPECA**.



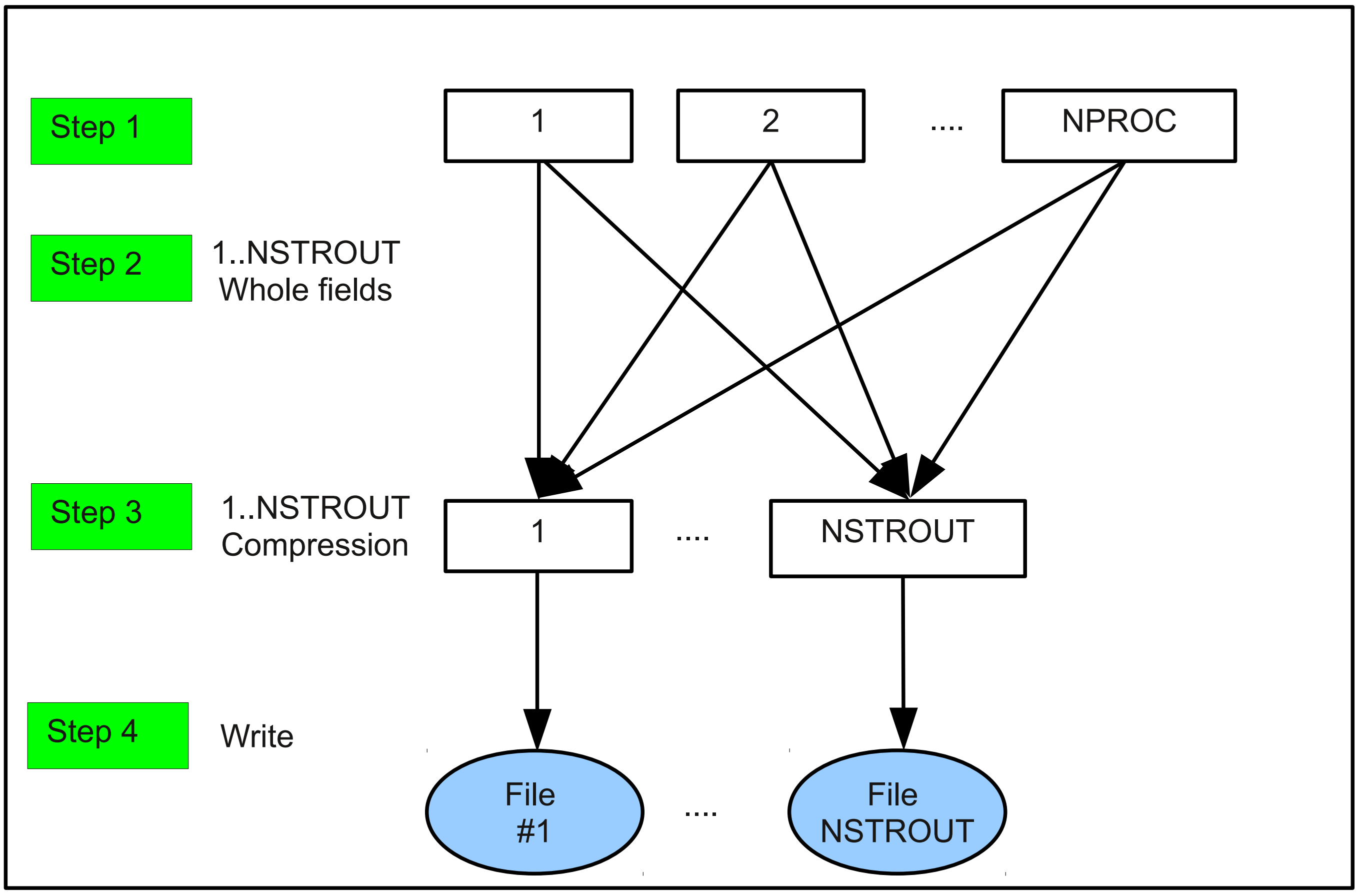
5 Extended traditional output

Output is still handled by model tasks. However, different methods have been implemented.

5.1 One file per NSTROUT task

```
&NAMPAR1
  NDISTIO(1)=1
/
```

A single file per `NSTROUT` processor is created. File have the following names : `ICMSH000+NNNN_MYPROC`.

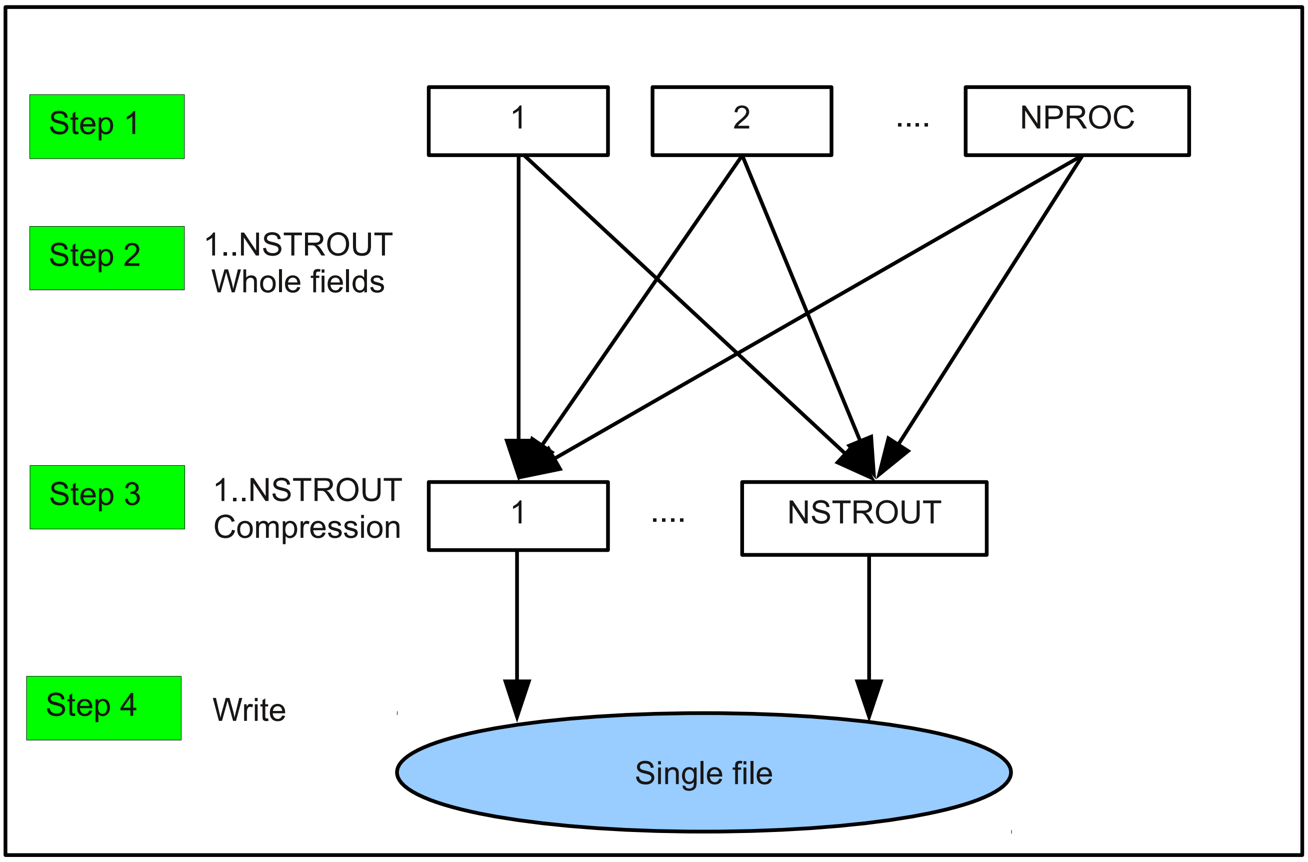


5.2 A single file written by all tasks

Yes, this is possible, but under the following conditions:

- **either** the file is being created,
- **or** it already exists, **and** fields to be written already exist with the right size
- **and** the number of fields to be written does not exceed the number of slots available in the primary FA index (usually 3072). The size of the FA primary index can be extended using a bigger "facteur multiplicatif".

Note that a MPI_ALLGATHERV is necessary, so that all NSTROUT tasks exchange field sizes and be able to compute offsets. Note eventually that writing compressed data does not go through the FA library, but through a C layer.



Writing to the same file by all tasks is enabled by the following parameter :

```
&NAMPAR1
  NDISTIO(1)=3,
/
```

5.3 *An alternative to GATHFLNM*

GATHFLNM is used to send/receive compressed fields from NSTROUT tasks to MPI #1. MPI sends are done in synchronous mode, in order not to flood MPI #1 with messages. GATHFLNM encodes/decodes data in a private buffer.

An alternative method with no intermediate buffer has been coded. It is activated by setting NDISTIO(5) to 1.

Setting NDISTIO(7) to 1 will enable “out of order message reception” by MPI #1 (default is to receive messages from tasks 1 to NSTROUT consecutively).

6 Disabling traditional output

For testing purposes, it is possible to disable the IO at different stages. Below are listed namelist parameters from NAMPAR1, and their effects :

	Field gathering	Field compression	Writing
`NDISTIO(2)=0` `NDISTIO(3)=0` `NDISTIO(4)=0`	Yes	Yes	Yes
`NDISTIO(2)=1` `NDISTIO(3)=0` `NDISTIO(4)=0`	Yes	Yes	No
`NDISTIO(2)=1` `NDISTIO(3)=1` `NDISTIO(4)=0`	Yes	No	Yes
`NDISTIO(2)=1` `NDISTIO(3)=1` `NDISTIO(4)=1`	No	No	No

7 Compacting fields with OpenMP

Its possible to compact fields with the FA library and OpenMP. This is done in **`WRSPECA`** and **`WRGP2FA`**. Activating FA compression under OpenMP requires setting **`NDISTIO(6)`** to 1.

Using FA with OpenMP requires :

- using the thread-safe interface of FA; all FA routines require an extra argument (see **`FACILE_MT`** in **`facile.F90`**)
- duplicating FA state for each thread (see **`wrmlppa.F90`** and **`fadup_mod.F`**).

8 SURFEX IO

SURFEX IO based on FA files is described here.

8.1 *Anatomy of a SURFEX historic file in FA*

A FA SURFEX historic file contains fields like a regular historic FA file. However, it contains additional information:

- non field data (integers, character strings, dates, logicals, etc...)
- a field index stored in 5 LFI records :
    - **`_FBUF_SIZE`**
    - **`_FBUF_DIM1`**
    - **`_FBUF_DIM2`**

- **_FBUF_NAME**
- **_FBUF_TYPE**

One-layer fields are prefixed with “SFX”; multi-layer fields are prefixed with “X”.

The field index is necessary to read data. SURFEX does not give a list of fields with their types to be read, this is why we need this index.

Eventually, note that fields are saved with the extension zone included.

8.2 Conversion from/to LFI

It is possible to perform such a conversion. Dedicated tools have been written and tested for AROME :

- `sfxtools`/`sfxfa2lfi`, conversion article by article
- `sfxtools`/`sfxlfi2fa`, conversion article by article; empty output file is required (for geometry information), and may be created using `lfitools`/`faempty`
- `sfxtools`/`sfxconv` converts a PGD and a SURFEX historic file from/to FA/LFI, going through SURFEX : SURFEX reads the data and writes it. The consequence is that the SURFEX version of the output file may be different from the input file.

  Beware that this program is not distributed, and has to load the whole SURFEX data into memory. Therefore, it may not work for large grids.

Note `sfxfa2lfi` and `sfxlfi2fa` rely on a module named `sfxflddesc_mod.F90` which contain the definition of SURFEX fields; however some (newly introduced) fields may not be present in this list. It is possible to pass the missing information to `sfxfa2lfi` and `sfxlfi2fa` using the option `-sfx-fld-desc.`

See embedded documentation (`sfxtools sfxfa2lfi --help`) for more details.

8.3 Modifying SURFEX historic files

Since the field index has to be updated accordingly when creating or deleting fields, a special tool has been developed. Its name is `sfxfilter`.

Modifying existing fields can be done using the regular FA/LFI interface, but adding/deleting fields requires `sfxtools/sfxfilter`; its usage is as follows :

```
$ SFXTOOLS SFXFILTER input-file output-file namelist
```

Fields are processed one by one, so memory consumption is low. The namelist syntax is :

```
&NAMSFXFILTER
  CDNOMA(1)="-ALBNIR_ISBA", ! Delete field ALBNIR_ISBA
  CDNOMA(2)="+ALBNIR_SOIL", ! Add field ALBNIR_SOIL
  CDTYPA(2)="X1",            ! ALBNIR_SOIL is a X1
  CDNOMA(3)="+LTOTO",        ! Add field LTOTO
  CDTYPA(3)="L1",            ! LTOTO is a L1
  KDIMSA(1,3)=10,            ! Dimensions of LTOTO
/
```

Note that fields are created but not initialized. Once created, they have to be initialized using regular FA/LFI API (like FAIENC, etc...).

8.4 Enabling FA

The following namelist parameters (`NAMPHMSE`/`YOMMSE`) are available :

Namelist parameter	Default	Comment
`LFMREAD`	.TRUE.	Read from LFI; setting this to .FALSE. enables reading from FA
`LFMWRIT`	.TRUE.	Write to LFI; setting this parameter to .FALSE. Enables writing to FA
`LPGDFWR`	.FALSE.	Write PGD fields
`LHISFWR`	.TRUE.	Write historic data
`LFTZERO`	.TRUE.	Set extension zone of SURFEX fields to zero after reading them.

File names are different and depend upon data encoding being FA or LFI :

		LFI	FA
PGD data	Input	PGD.lfi	Const.Clim.sfx
SURFEX initial conditions	Input	TEST.lfi	ICMSH0000INIT.sfx
Historic data	Output	AROMOUT_0003.lfi	ICMSH0000+003.sfx

8.5 The SURFEX cache

The SURFEX field cache is a big structure where all SURFEX fields can be buffered before initializing SURFEX (during the set-up), or before producing an output file. It is implemented in `modd_io_surf_aro.F90`, and may contain fields of the following types:

Code	Prefix	Description	Type
T0	SFX.	Scalar date	metadata
T1	SFX.	1D-array of dates	metadata

N0	SFX.	Scalar integer	metadata
N1	SFX.	1D-array of integers	metadata
C0	SFX.	Scalar character string	metadata
X0	SFX.	Scalar real	metadata
L0	SFX.	Scalar logical	metadata
L1	SFX.	1D-array of logicals	metadata
X1	SFX.	2D SURFEX 1-level field	data
X2	X	2D SURFEX field with several levels	data

Note that all fields but those of type X1 and X2 are meta-data and are expected to have a very small size. Only X1 and X2 are real numeric data.

Two variables of type `SURFEX_FIELD_BUF_CACHE` are defined :

- YSURFEX_CACHE_IN; used for reading fields; see `aroini_surfa.F90`, `aroini_surfb.F90`, `aroini_surfc.F90` and `sugridsfx.F90`
- YSURFEX_CACHE_OUT; used for producing historic files; see `aro_surf_diag.F90` and `wrsfx.F90`

This object is complicated, but we list anyway some of the possible operations (all should be prefixed with `SURFEX_FIELD_BUF`) :

		Description
`..._ADD`	generic	Add a new field
`..._SET`	generic	Set a new field (not for X1/X2)
`..._GET`	generic	Get a new field value (except for X1/2), get a pointer on field (X1/2)
`..._PREALLOC`	subroutine	Allocate big arrays for X1/X2 fields
`..._GET2DF`	subroutine	Get pointers on X1/X2 fields and their meta-data
`..._READ_MISC`	subroutine	Read meta-data (fields different from X1/X2)
`..._WRITE_MISC`	subroutine	Write meta-data (fields different from X1/X2)
`..._DEALLOC`	subroutine	Deallocate big arrays (X1/X2)
`..._EXIST`	function	

These routines are called from `(read|write)_surf(T0|T1|X0|X1|X2|...)_aro.F90` routines.

9 The IO server

The IO server's purpose is to offload IO (only output for now) to a group of dedicated MPI tasks. The code used to do that is located in `arp/io_serv` ; the amount of code involved (comments and blank lines included) is about 5000 lines.

The IO server can handle historic fields (grid-point and spectral), and fields produced by Fullpos. SURFEX output can be redirected to the IO server as well.

The IO server can receive and process whole fields and field fragments, in which case whole fields are re-created by the IO server itself.

Fields are compressed using the FA library, and written to disk in the FA format. The result is supposed to be bitwise reproducible when compared to traditional output.

9.1 Implementation of the IO server

Handling of asynchronous "sends"

All operations described in this section relate to the model side of the IO server; that is, these operations are carried out by model MPI tasks.

Asynchronous sends in an MPI context are performed using `MPI_ISEND` (wrapped in `MPL_SEND`); the major issue in this context is memory management : since memory cannot be reclaimed as soon as the `MPI_ISEND` returns, the IO server must manage memory itself.

Hence memory buffers used for communication are allocated by the IO server, and references to these buffers are kept in a list, with the associated send request IDs. The number of memory buffers is limited (maximum number specified in namelist); once the number of allocated buffers has reached its maximum value, model MPI tasks have to wait in order to allocate memory (dedicated to asynchronous communications for IO).

The list of pending asynchronous sends is periodically polled (each time some memory is needed for new sends) :

- when the number of allocated buffers has reached its maximum value, `MPI_WAITSOME` is invoked; the MPI task has then to wait until at least one of already posted send requests have completed.
- when the number of allocated buffers has not reached its maximum value, then `MPI_TESTSOME` is used; this subroutine returns immediately. Buffers whose requests have completed are deallocated.

Allocation of a new buffer is handled by `IO_SERV_ALLOC_BUF`; this routine invokes `IO_SERV_RECLAIM_BUF_SPACE` in either waiting or no waiting mode.

Eventually, `IO_SERV_RECLAIM_BUF_SPACE` is invoked by `IO_SERV_FLUSH` when the model exits until all pending MPI requests have completed (and all associated buffers have been freed).

The routine used to send data to the IO server is `IO_SERV_SEND` ; it is currently used in `WRGP2FAFP`, `WRSPECA` and `WRGP2FA` .

IO server message "receives"

All operations described in this paragraph take place on the IO server side.

The IO server does not assume that the messages arrive in a specific order; the IO server enters an endless loop and probes message arrival using `MPI_PROBE` ; once a message is known to be available, a buffer of the corresponding size is allocated and the message is read.

The message is then posted to a thread-safe queue (FIFO) in order to be post-processed. The IO server message reception runs in subroutine `IO_SERV_RUN` .

IO server termination

The IO server exits its endless loop when it receives stop messages; for now, each IO server MPI task waits until it have received a stop message from each model MPI task.

Multi-threading

The IO server can run in multi-threading mode. As this has not proved to be efficient yet, it is not documented here.

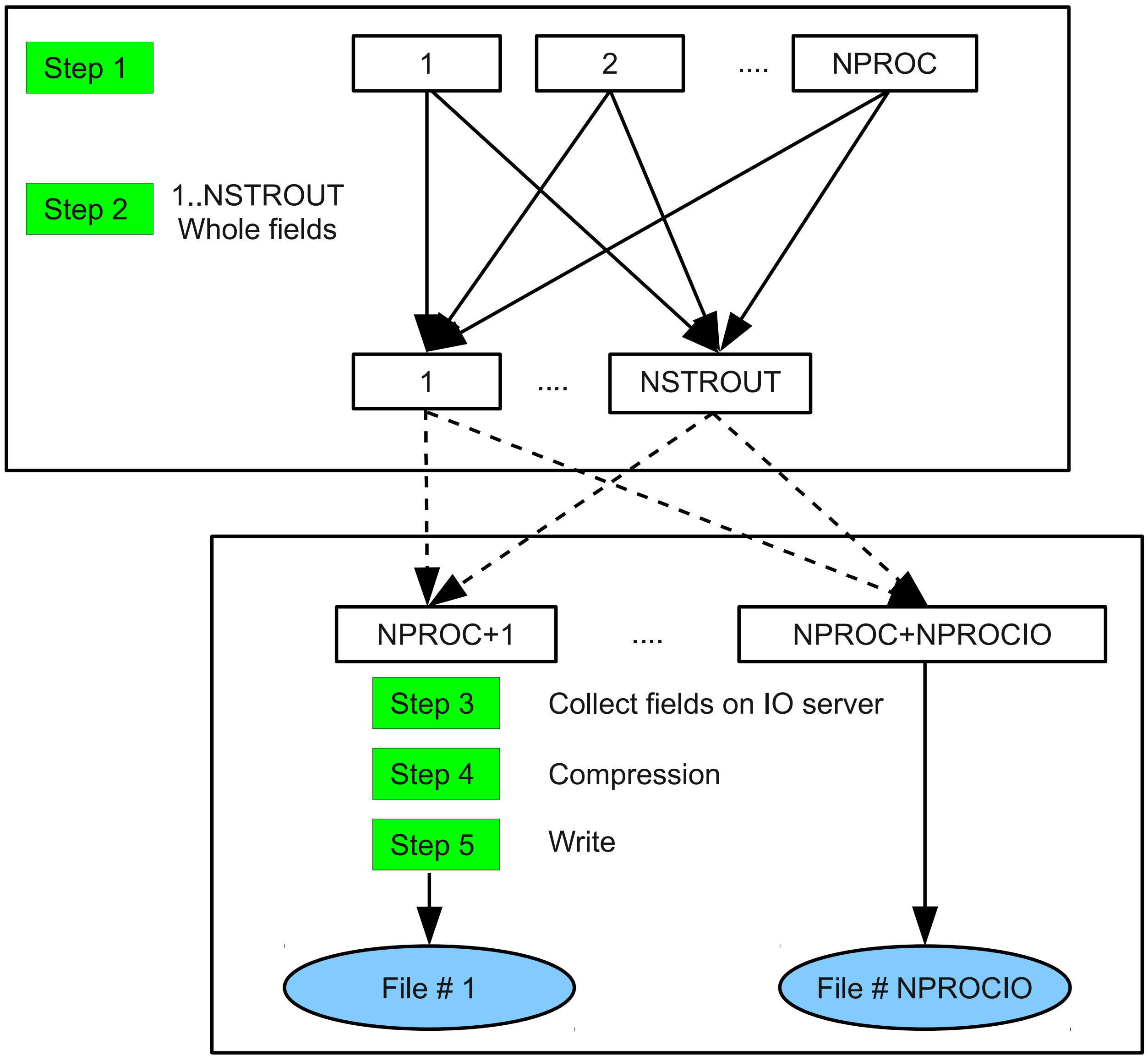
9.2 Namelist parameters

Namelist parameter	Comment
`NPROC_IO`	Number of MPI tasks involved in the IO server
`NIO_SERV_METHOD`	Synchronous (1), asynchronous (2)
`NIO_SERV_BUF_MAXSIZE`	Maximum number of send buffers allocated on a single processor
`LMSG_FLUSH_CLIENT`	Flush client log output
`LMSG_FLUSH_SERVER`	Flush server log output
`LCOMPRESS_FA`	Apply FA compression
`LDUMPNORMS`	Dumps norms (`wrgp2fa.F90`)
`NMSG_LEVEL_CLIENT`	Log level for client tasks; from 0 (silent) to 3 (verbose)
`NMSG_LEVEL_SERVER`	Log level for server tasks; from 0 (silent) to 3 (verbose)
`NOUTPUT_FMT`	0 = binary, 1 = FA
`LUSE_MAP`	.FALSE. = field based .TRUE. = gather based
`NPROCESS_LEVEL`	processing level (for testing, debugging):

	• `NIO_SERV_PROCESS_NONE = 0` create IO server, but do nothing • `NIO_SERV_PROCESS_RECV = 1` receive messages `NIO_SERV_PROCESS_RECO = 2` pass fields to compress threads `NIO_SERV_PROCESS_COMP = 3` compress threads `NIO_SERV_PROCESS_COWR = 4` pass compressed data to writer threads `NIO_SERV_PROCESS_WRIT = 5` write data

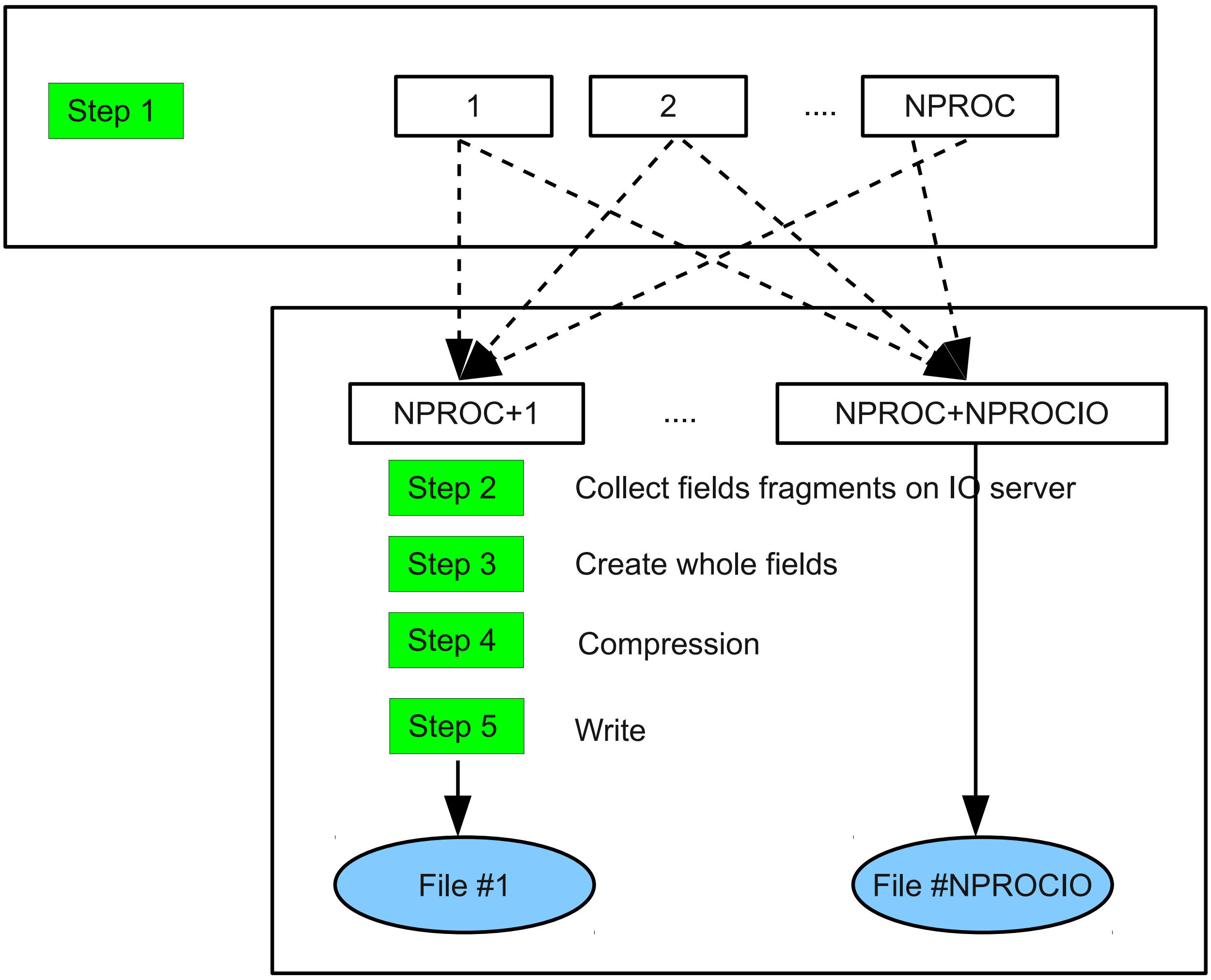
9.3 Field based mode

In this mode, model tasks still gather field data on **`NSTROUT`** MPI tasks, but whole fields are send to the IO server tasks in asynchronous mode, in order to avoid the cost of extra buffering.



9.4 Gather based mode

In this mode, field fragments are sent to the IO server, which collects them in queues and re-create whole fields. Then all goes on as usual (compression + writing). Note that NSTROUT does not play any role in this mode.



9.5 Reading data produced by the IO server

The IO server creates a single FA file per IO task. Reading data back with the model requires some attention.

Using facat

`lfitools`/`facat` makes it possible to concatenate several FA files with the same geometry into a single one. The model then reads the result of the concatenation as usual.

Using an index file

The model can read input data spread across multiple files. It is first necessary to create an index using `lfitools`/`faidx` whose usage is :

```
$ lfitools faidx file1 file2 ... fileN file.idx
```

Once this index is created, setting the following namelist parameter will tell the model to try to look for the index file, read it and set-up things so that it open the right files to read the required fields :

```
&NAMCT0
  LIOTRYIDX=.TRUE.
/
```

Since input and output goes through the same routines for making field lists (for instance iogrida_mod.F90, etc...), it should not be necessary for all model tasks to open all files produced by the IO server. If this were to happen and were a problem, it should in principle be possible to re-organize the code a little bit to avoid multiple open/close operations.

10 Fullpos & SURFEX

The creation of SURFEX input files from ARPEGE/ISBA files has been distributed. `fp2sx1fa.F90` calls `PREP` bits of code in a distributed context. Another routine (`rdclimosfx.F90`) has been written to read SURFEX PGD (`Const.Clim.sfx`).

Beware, though, that this works only with the FA format.

11 Further developments

11.1 Reading input files with Multi-Threading

This should be possible very soon. For now, FA/LFI have a thread-safe interface, but reading/writing files is not thread-safe in Fortran. In principle, it should be enough to replace Fortran IO with C IO.

This would make it possible to use multiple threads to read a FA/LFI file.

11.2 Simplifying the code

Many options are available for doing IO. This is probably too much to maintain, and this will be simplified when enough experimentation has been conducted on scalar machines.

11.3 Removing LFI from mse project

This has to happen someday when everybody in the ARPEGE, ALADIN, AROME, HARMONIE world use FA for storing SURFEX fields. The LFI layer coded in mse is a terrible mess and should be removed.