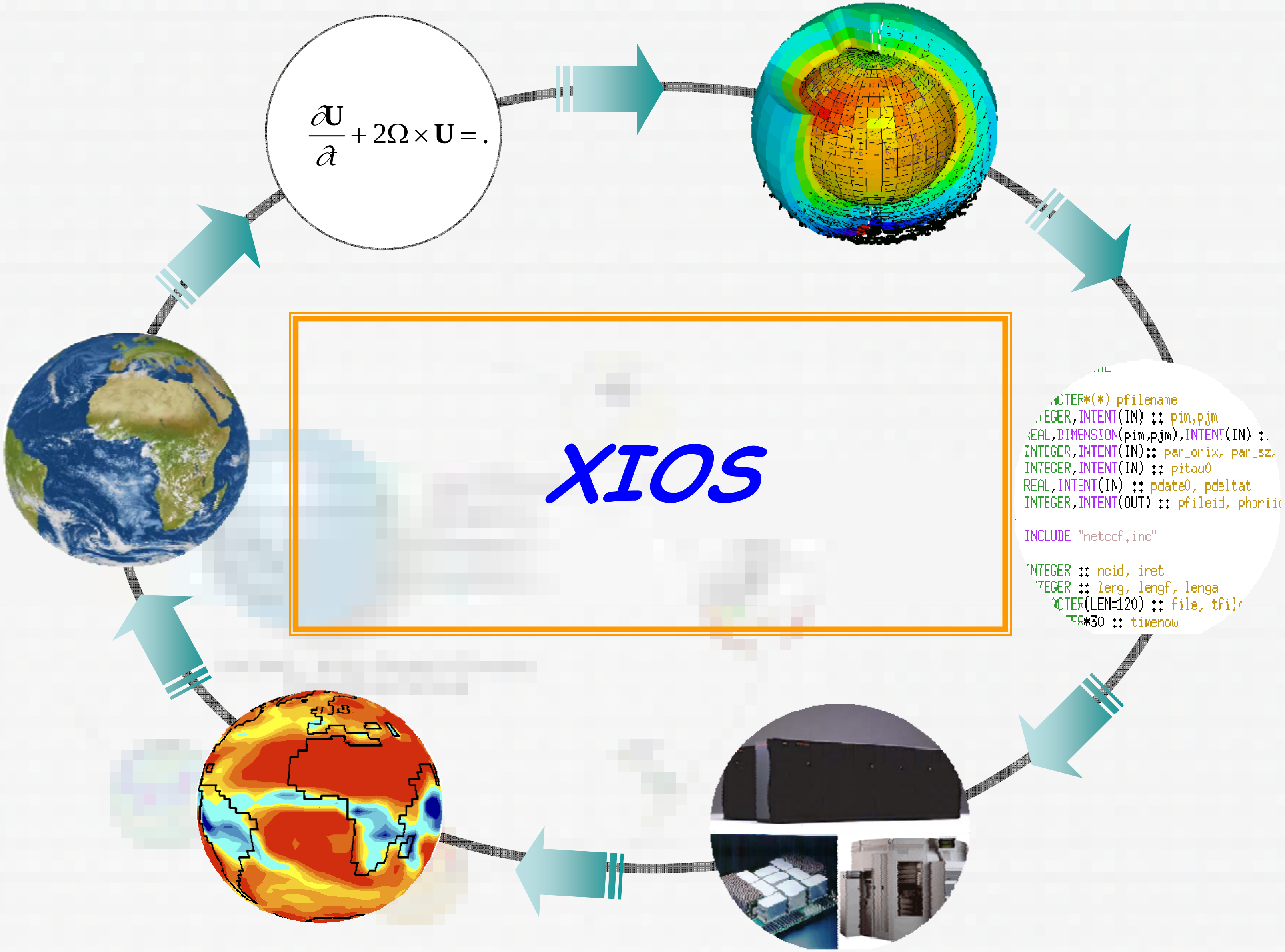
Yann Meurdesoif(LSCE-IPSL), **H. Ozdoba**, A. Caubel, O. Marti

XIOS



XIOS - Motivation

- XIOS stands for XML – IO – SERVER
  - Library dedicated to IO management of climate code.
    - management of output diagnostic, history file.
    - Temporal post-processing operation (averaging, max/min, instant, etc...)
    - Spatial post-processing operation.
- Motivation
  - Before : IOIPSL : output library for the IPSL model.
    - Enable management of output file in netcdf format.
    - Management of calendar, restart file and history diagnostics.
    - Management of temporal operation like averaging.
  - Good tool but suffer of several drawback
    - Not very flexible to use.
    - Need to recompile for each modification on IO definitions.
    - Many call parameters for IO write subroutine. Even more for definition phase.
    - A lot of unnecessary repeated parameters.
    - Need to conserve a lot of handle for IO calling.
      - Concentration of IO call in the same part of the code
    - No management of parallelism or multithreading
    - 1 file by computing processes, file need to be rebuild in post-processing phase
    - Loss of efficiency for great number of computing core, for output and for rebuild.

XIOS in summary…

- XIOS aims to solve these problems with 2 main goals :
  - Flexibility
    - Simplification of the IO management into the code
      - Minimize calling subroutine related to IO definition (file creation, axis and dimensions management, adding and output field…)
      - Minimize argument of IO call.
    - Ideally : output a field require only a identifier and the data.
      - `CALL send_field("field_id", field)`
    - Outsourcing the output definition in an XML file
      - Hierarchical management of definition with inheritance concept
      - Simple and more compact definition
      - Avoid unnecessary repetition
    - Changing IO definitions without recompiling
      - Everything is dynamic, XML file is parsed at runtime.
  - Performance
    - Writing data must not slow down the computation.
      - Simultaneous writing and computing by asynchronous call.

XIOS - Historical review

- Using one or more "server" processes dedicated exclusively to the IO management.
  - Asynchronous transfer of data from clients to servers.
  - Asynchronous data writing by each server.
- Use of parallel file system ability via Netcdf4-HDF5 file format.
  - Simultaneous writing in a same single file by all servers
  - No more post-processing rebuilding of the files

Historical review

End 2009 : « Proof of concept » : XMLIO-SERVER-V0

- Written in Fortran 90
- External description of IO definition in an XML file
- Implements server functionality.
- But still using the old IOIPSL layer on back-end.
  - no management of parallelism, 1 file by server needed to be rebuild.
- Mid-2010 : integration of XMLIO-SERVER into the official release of NEMO.

Mi-2010 – end 2011 : Complete rewriting in C++

- Funded as part of IS-ENES (H. Ozdoba, 18 months)
- C++ required for using object-oriented programming.
- Interoperability C/C++/Fortran through Fortran 2003 normalization feature.
- Remove the old IOIPSL layer.
- Improved functionality and performance

- Parallel IO management
  - No more rebuilding phase
- XMLIO-SERVER becomes XIOS.
- End 2011 : first alpha release : XIOS-V1.0-alpha1.
- XIOS integration into NEMO and testing.
- February 2012 : second alpha release : XIOS-V1.0-alpha2.
- Now : ~ 25000 code lines under SVN

To extract and install :

- launch_xios script :

```
#!/bin/bash
svn export http://forge.ipsl.jussieu.fr/ioserver/svn/XIOS/extract_xios
./extract_xios $*
```

```
> launch_xios --interactive
```

- Use FCM (developed at MetOffice) to build dependency and compile.
- Tested on intel (ifort/icc) and gnu (gfortran/g++) compiler

XIOS - HELLO WORD

- A simplest application with XIOS : hello word !
  - output field : field_A in the file output.nc

```xml
<?xml version="1.0"?>
<simulation>
  <context id="hello_word" calendar_type="Gregorian" start_date="2012-02-27 15:00:00">

    <axis_definition>
      <axis id="axis_A" value="1.0" size="1" />
    </axis_definition>

    <domain_definition>
      <domain id="domain_A" />
    </domain_definition>

    <grid_definition>
      <grid id="grid_A" domain_ref="domain_A"  axis_ref="axis_A" />
    </grid_definition>

    <field_definition >
      <field id="field_A"  operation="average" freq_op="1h" grid_ref="grid_A" />
    </field_definition>
```

Hello Word - XML side

```
    <file_definition type="one_file" output_freq="1d" enabled=".TRUE.">
       <file id="output" name="output">
          <field field_ref="field_A" />
       </file>
     </file_definition>
  </context>

  <context id="xios">
      <variable_definition>
        <variable_group id="parameters" >
          <variable id="using_server" type="boolean">true</variable>
        </variable_group>
      </variable_definition>
  </context>

</simulation>
```

Hello Word - Fortran side

```
PROGRAM test_cs
IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER :: rank
  INTEGER :: size
  INTEGER :: ierr

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)

  IF (rank<3) THEN
   CALL client(rank,3)
  ELSE
    CALL server
  ENDIF

  CALL MPI_FINALIZE(ierr)
END PROGRAM test_cs

SUBROUTINE server
  USE xios
  IMPLICIT NONE
    CALL xios_init_server
END SUBROUTINE server
```

Hello Word - Fortran side

```
SUBROUTINE client(rank,size)
  USE xios
  IMPLICIT NONE
  INTEGER :: rank, size
  TYPE(xios_time)      :: dtime
  DOUBLE PRECISION,ALLOCATABLE :: lon(:,:),lat(:,:),field_A(:,:)
  ! other variable declaration and initialisation
  ! .........
  CALL xios_initialize("client", return_comm=comm)

  CALL xios_context_initialize("hello_word",comm)
  CALL xios_set_current_context("hello_word")

  CALL xios_set_domain_attr("domain_A",ni_glo=ni_glo, nj_glo=nj_glo,     &
                            ibegin=ibegin, ni=ni,jbegin=jbegin,nj=nj)
  CALL xios_set_domain_attr("domain_A",lonvalue=RESHAPE(lon,(/ni*nj/)),  &
                            latvalue=RESHAPE(lat,(/ni*nj/)))
  dtime%second=3600
  CALL xios_set_timestep(dtime)

  CALL xios_close_context_definition()
```

Hello Word - Fortran side

```
! time loop

  DO ts=1,96
    CALL xios_update_calendar(ts)
    CALL xios_send_field("field_A",field_A)
  ENDDO

  CALL xios_context_finalize()
  CALL xios_finalize()

END SUBROUTINE client
```

XIOS - XML : generality

- Different family of element
  - context, axis, domain, grid, field, file and variable.

- Each family has three flavor (except for context)
  - declaration of the root element : ie : `<file_definition />`
    - can contain element groups or elements
  - declaration of a group element : ie : `<file_group />`
    - can contain element groups or elements
  - declaration of an element : ie : `<file />`

- Each element may have several attributes
    - ie : `<file id="out" name="output" output_freq="1d" />`
  - Attributes give information to the related element
  - Some attributes are mandatory, so error is generated without assigned value
  - Some other are optional but have a default value
  - Some other are completely optional
  - Special attribute **id** : identifier of the element
    - used to take a reference of the corresponding element
    - must be unique for a kind of element
    - is optional, but no reference to the corresponding element can be done later

XIOS - XML : generality

- XIOS-XML has a based tree structure.
  - Parent-child oriented relation

- Each operation written in XML file may be done from the Fortran interface
  - Create or adding an element in the XML tree

```
CALL xios_get_handle("field_definition", field_group_handle)
CALL xios_add_child(field_group_handle,field_handle,id="toce")
```

  - Complete or define attributes of an element
    - Using handle

```
CALL xios_set_field_attribut(field_handle,long_name="Temperature", unit="degC")
```

    - Or using id

```
CALL xios_set_field_attribut(id="toce", enabled=.TRUE.)
```

  - Query an attribute value from xml file

```
CALL xios_get_field_attribut(id="toce", enabled=is_enabled)
```

XIOS - XML : generality

- XML file can be split in different parts.
  - Very useful to preserve model independency, i.e. for coupled model
  - Using attribute "src" in context, group or definition element

```
File iodef.xml :

<context src="./nemo_def.xml" />

file nemo_def.xml :

<context id="nemo" calendar_type="Gregorian" start_date="01-01-2000 00:00:00">
...
...
</context>
```

Inheritance mechanism

Grouping an inheritance

- All children inherit attributes from parent.
- An attribute defined in a child replace the inherited attribute value.
  - Avoid unnecessary repetition of attribute declaration
- Special attribute “id” is never inherited

```
<field_definition level="1" prec="4" operation="average" enabled=".TRUE.">

  <field_group id="grid_V"   domain_ref="grid_V">
    <field id="vtau"   long_name="Wind Stress along j-axis"    unit="N/m2" enabled=".FALSE."/>
    <field id="voce"   long_name="ocean current along j-axis"  unit="m/s" axis_ref="depthv" />
  </field_group>

  <field_group id="grid_W" domain_ref="grid_W">

    <field_group axis_ref="depthw">
      <field id="woce"      long_name="ocean vertical velocity"           unit="m/s" />
      <field id="woce_eff"  long_name="effective ocean vertical velocity" unit="m/s" />
    </field_group>

    <field id="aht2d"       long_name="lateral eddy diffusivity"          unit="m2/s" />

  </field group>
</field_definition>
```

Inheritance mechanism

Inheritance by reference

- Reference bind current object to the referenced object.
- If the referenced object is of the same type, current object inherits of all its attributes.

```
<field id="toce" long_name="temperature (Celcius)" unit="degC" grid_ref="Grid_T" />
<field id="toce_K"  field_ref="toce" long_name="temperature (Kelvin)"  unit="degK" />
```

- “field_group” referencing include all fields child in the current group.

```
<field_definition/>
  <field_group id="grid_T" domain_ref="grid_T">
    <field id="toce"   long_name="temperature" unit="degC" axis_ref="deptht"   />
    <field id="soce"   long_name="salinity"    unit="psu"  axis_ref="deptht"   />
    <field id="sst"    long_name="sea surface temperature" unit="degC"         />
    <field id="sst2"   long_name="square of sea surface temperature"  unit="degC2" />
</field_group>
</field_definition>

<file_definition>
   <file id="1d" name="out_1day" output_freq="1day" enabled=".TRUE." />
     <field_group field_group_ref="grid_T" />
   </file>
</file_definition>
```

Context and calendar

Context : <context />

- Context are useful to isolate IO definition from different code or part of a code
  - ie : IO definition can be done independently between different code of a coupled model
- No interference is possible between 2 different contexts
  - Unique Id can be reused in different contexts.
- Each context has it own calendar and an associated timestep
  - timestep is the heartbeat of a context

Calendar

- XIOS can manage different calendar with context attribute "**calendar_type**"
  - **Gregorian**
  - **D360**
  - **NoLeap**
  - **AllLeap**
  - **Julian**
- Date Format : ie : "**2012-02-27 15:30:00**"

Duration

- Can manage different units
  - year : y
  - month : mo
  - day : d
  - hour : h
  - minute : mi
  - second : s
- Value of unit may be integer or floating (not recommended), mixed unit may be used in a duration definition
  - ie : "**1mo2d1.5h30s**"
- A duration depend of the calendar for year, month and day value.
  - "**2012-02-27 15:30:00**" + "1 mo" => "**2012-03-27 15:30:00**"

Grids definition

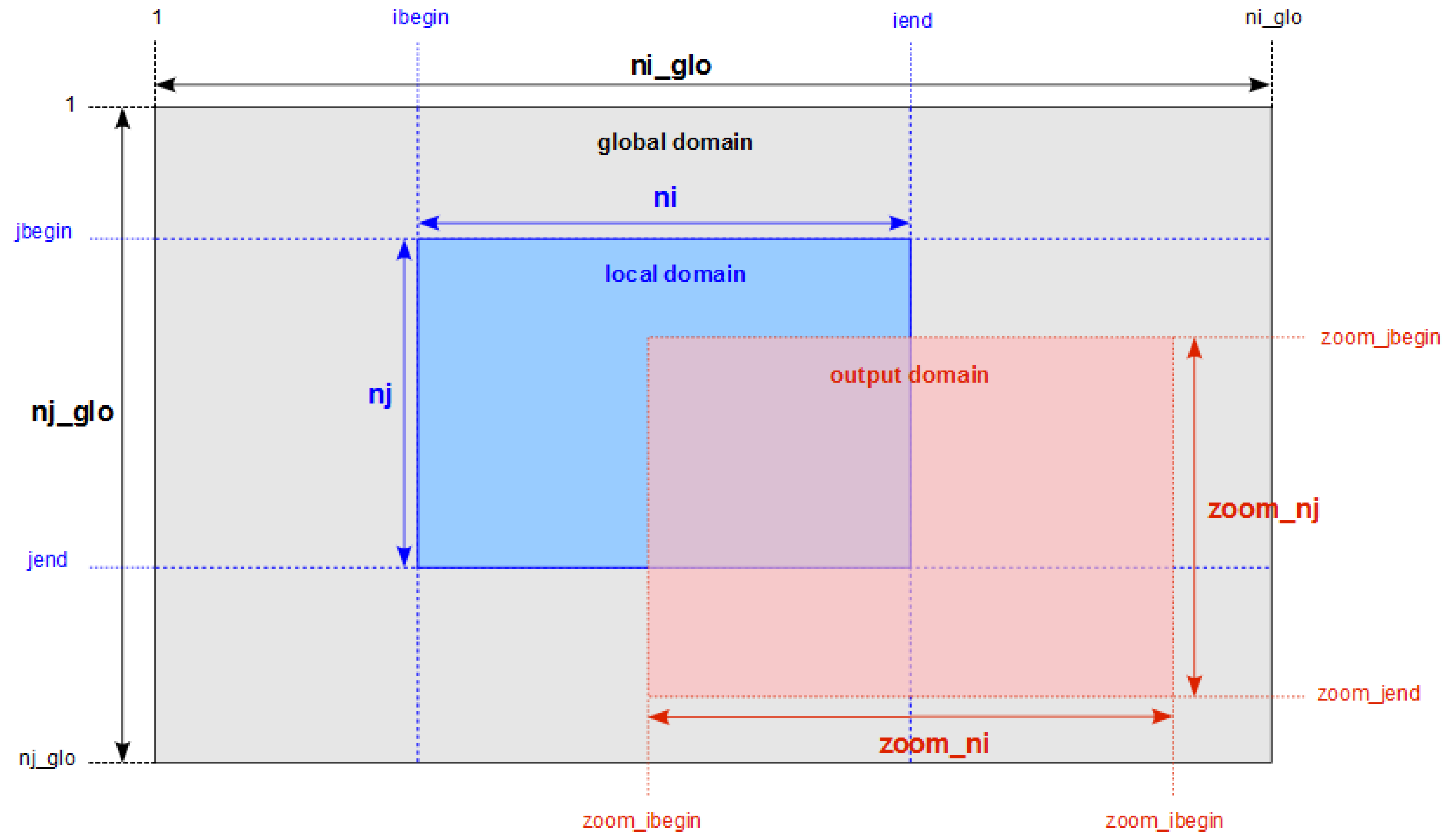
- Grid : **<grid />**
  - Only Cartesian or curvilinear grid can be manage today by XIOS
  - A grid is defining by association (referencing) of an horizontal domain and optionally a vertical axis (3D grid otherwise 2D horizontal grid)
    - `<grid id="grid_A" domain_ref="domain_A"  axis_ref="axis_A" />`

- **Vertical axis : <axis />**
  - Can be defining with attributes : **size**, **value** and **unit.**

- Horizontal domain : <domain />
  - Horizontal layer is considered to be distributed between the different processes.
  - 2D global domain is the domain that will be output in a file.
  - 2D local domain is the domain owned by one process (within MPI meaning)
  - Global attributes :
    - **ni_glo, nj_glo** : dimension of the global grid
    - **zoom_ibegin, zoom_ni, zoom_jbegin, zoom_nj** : define zooming functionality : only a part of the global domain will be output. Default zoom is global domain

Grids definition

- Local attributes : define the local grid in connection with the global grid
  - **ibegin, ni, [iend]**
  - **jbegin, nj, [jend]**



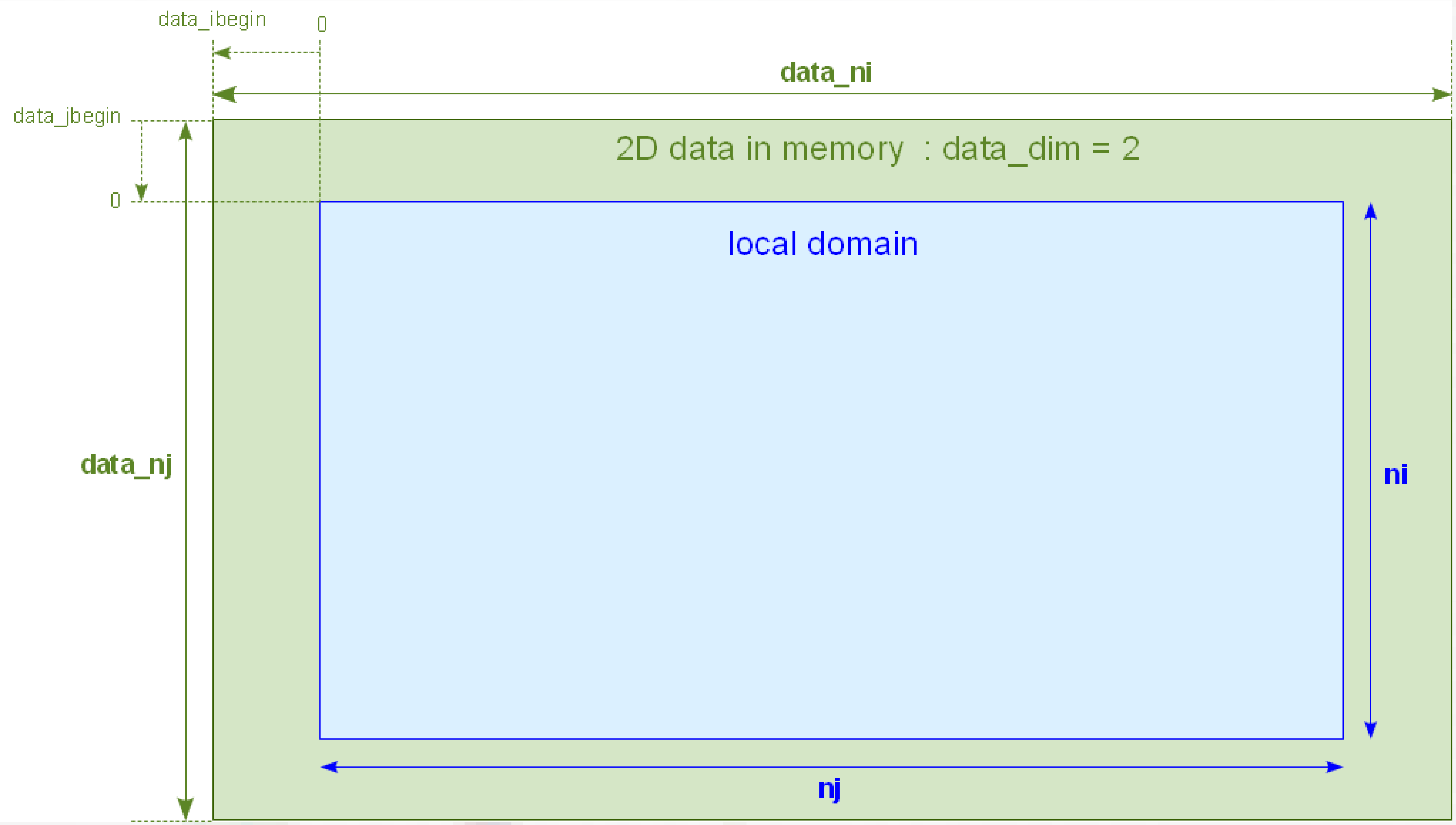
Grids definition

- XIOS need to know how the data of a field to be output are stored in the local process memory.
  - 1D ("**data_dim=1**") or 2D ("**data_dim=2**") field on horizontal domain may be described.
  - **data_ibegin** : offset in regard to **ibegin** local domain, for the first dimension
  - **data_ni** : size of the data for the first dimension
  - **data_jbegin, data_nj** : for the second dimension (if **data_dim=2**)

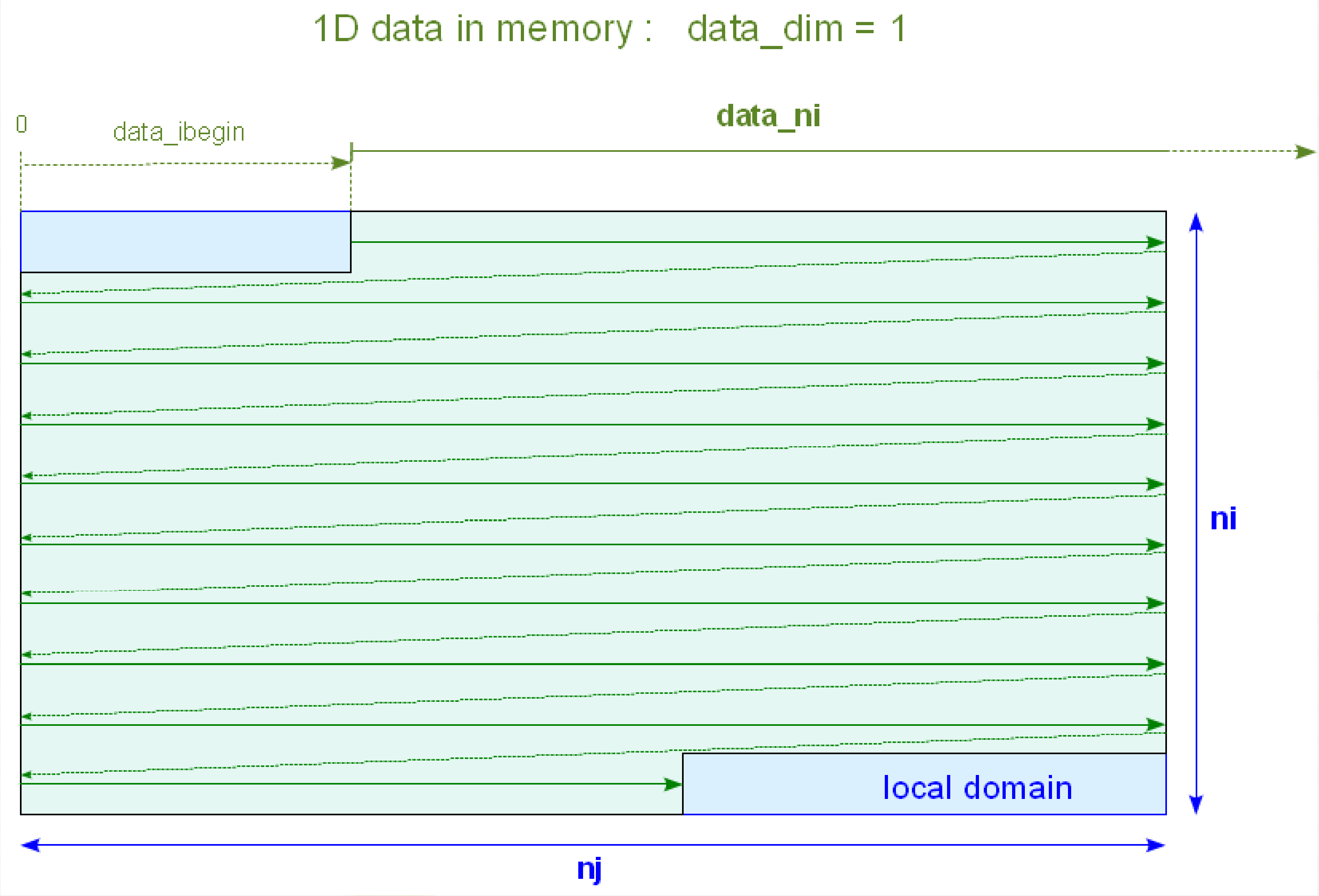
- By this way overlapping (ghost) cell can be take into account using negative offset.
  - XIOS will extract useful data from the array address.
  - default value are no overlapping cell
  - **data_ibegin=0, data_jbegin=0, data_ni=ni, data_nj=nj** : mapped to local domain

- Indexed grid (compressed), ie for land-point, may be described by adding index attribute :
  - **data_n_index** : size of the indexed data
  - **data_i_index** : array containing index for the first dimension
  - **data_j_index** : array containing index for the second dimension (if **data_dim=2**)

Grids definition



Grids definition



Field

Field : <field />

- Describe data of field to be output
- A field must be associated to a grid by attribute referencing :
  - grid_ref : field is associated to the referred grid
  - domain_ref : field is associated to the referred domain (2D field)
  - domain_ref and axis_ref : field is associated to a grid composed of the referred domain and axis (3D field).
- Field array dimension must be conform to whom described in the referred grid (data_dim, data_ni, data_nj).
- Field can be sent at each timestep through the fortran interface :

```
CALL xios_send_field("field_id",field)
```

- Temporal operation may be done by using field value given at each timestep : "**operation**" attribute :
  - **once** : field is output only the fisrt time
  - **instant** : instant value
  - **average** : temporal averaging on the output period
  - **minimum** : retain only minimum value
  - **maximum** : retain only maximum value.
- In case of time sub-splitting in the model, a freq_op attribut may be gaven
  - Extract field value only at freq_op (default freq_op = timestep)

Field

- **field_ref** attribute
  - inherit the attributes of the referred field
  - inherit the data value of the referred field

- Spatial operation between fields, scalar values and variables (soon implemented)

```
<field id="A" />
<field id="B" />
<field id="C" operation="average"> $A+$B </field>
<field id="D"> 1e3*($D/@C) </field>
```

  - $field : use the instantaneous value of a field.
  - @field : use field value after temporal operation.
  - Operation are performed on all grid point.
  - involved field must be defined one the same grid.

- Other main attribute :
  - **name** : field name which will appear in the ouput file. By default use "id" attribute.
  - **enabled** = "true/false": activate or deactivate the field
  - **prec** : (4 or 8) : float or double value are output in the file
  - **missing_value** : value for missing value
  - …

File

File : <file />

- Define an output file
- File can contain field_group and/or field child element.
- All field enclosed in a parent file are candidate to be output in the file if they are active. Better to use reference but not mandatory.
- Output frequency is given by “output_freq” attribut.
  - Temporal operations of the enclosed fields are applied on the output_freq period.

```
<file id="1d" name="out_1day" output_freq="1d" enabled=".TRUE.">
   <field field_ref="toce" operation="average" enabled=".FALSE." />
   <field name="max_toce" field_ref="toce" operation="maximum" />
 </file>
```

- Other main attribute
  - **name** : file name
  - **enabled** =true/false : activate or deactivate a file
  - **split_freq** : split file at a given frequency
  - …

Variable

- Variable : <variable />
  - Variables can be defined in each context and be queried through the fortran interface
  - Useful to set code parameters, can replace namelist usage with more flexibility.

```
<context id="xios">
  <variable_definition>

    <variable_group id="buffer">
       buffer_size = 1E6;
       buffer_server_factor_size = 2
    </variable_group>

    <variable_group id="coupling">
      <variable id="using_server" type="boolean">true</variable>
      <variable id="using_oasis"  type="boolean">false</variable>
    </variable_group>

  </variable_definition>
</context>
```

- `CALL xios_getin(“varid”,var)`

Communication layer

Client-Server functionality

Client-Server functionality

- Adding one or more XIOS server processes dedicated to writing data
- Client are MPI processes of the computing code

Why for ?

- Stages of writing are totally supported by servers ; client time computation is not affected by writing.
  - Writing and computing are done concurrently.
- Only server processes access to the file system :
  - Less file system solicitation
  - Better performance
- Load balancing
  - Add enough server to balance IO over computation ratio.
- Data transfert Client->Server are totally asynchronous :
  - Using non-blocking request MPI_ISend/MPI_IRecv/MPI_IProb.
  - Overlapping communication/computation.
  - No extra cost on client side related to interprocess data transfer.

Client-Server functionality

- Usage
  - XIOS can operate either in online mode, either in server mode.
    - switching parameter at runtime
    - using_server=true/false
- In online mode, client codes are linked with the XIOS library and perform themselves writing on disk.
    - May suffer of computation time penalty during writing.

- In server mode, client codes are interfaced with the XIOS library to send the data to the server processes.
  - Launching different MPI executable (MPMD mode)

```
mpirun -np 32 nemo.x -np 4 xios.x
```

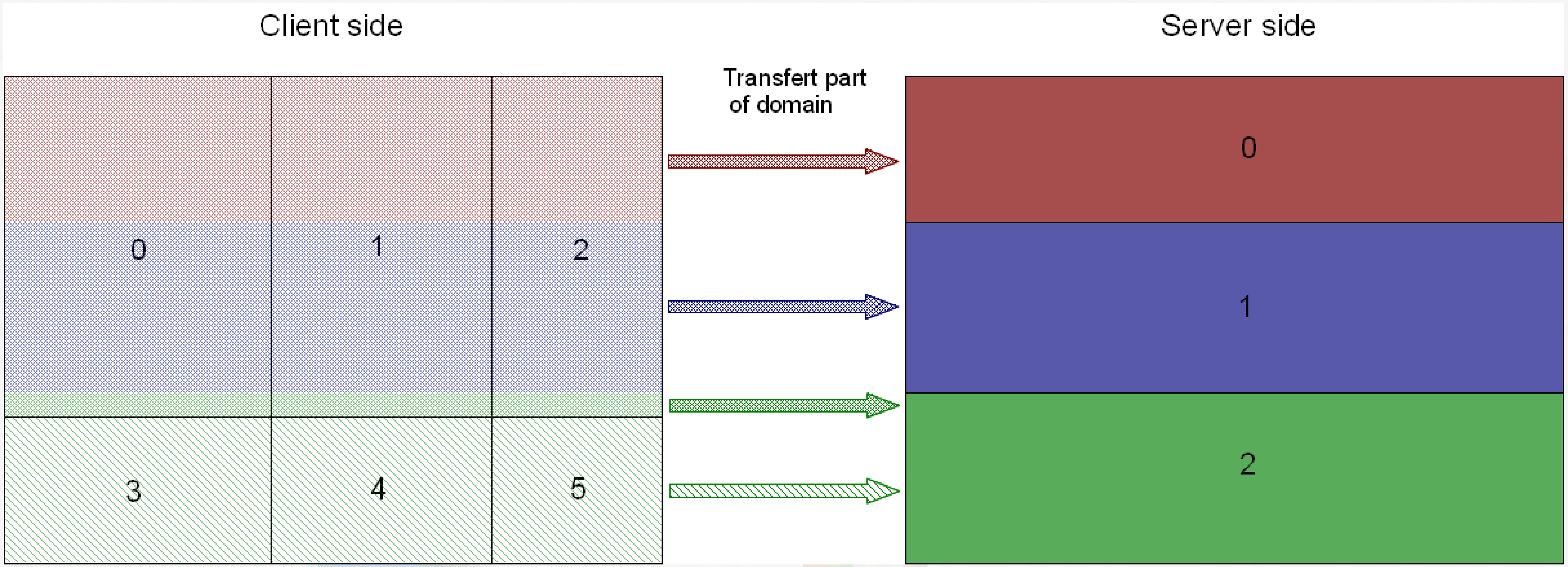
  - The temporal and spatial operation could be performed either
    - client side : data sent to the server only at writing time
    - server side : data sent to the server at each time step => many more communication.

Client-Server functionality

- XIOS library can manage MPI communicator distribution between client and server, either :
  - With its internal routines (model alone +XIOS)
  - By interfacing with the OASIS coupler (coupled model + XIOS)
  - ➧ switch parameter : using_oasis = true

Data Distribution

- Distribution of data field between client and server



  - Clients 0, 1, 2 send part of it domain to server 0, 1 and 2
  - Clients 3, 4, 5 send part of it domain only to server 2
- Distribution on client is managed by the code
- Distribution of data on server is equally distributed over the second dimension
- Client can communicate with several servers.
- Server can receive data from several clients.

Communicator & context registration

Communicator splitting

- Clients and XIOS server required to have their own communicator
- Global communicator may be split either by XIOS library or by using OASIS coupler.
  - Done during the client initialization phase, each client code is identified by a unique id.

    ```
    CALL xios_initialise("code_id", return_comm)
    ```
  - Call must be done by every process of all clients in MPI_COMM_WORLD communicator.
- A split communicator is returned.
- At this point, servers are initialized and are now listening for context registration.

Context registration

- A context is associated to a communicator.
- Before using a context, it must be registered, with its "id" and the corresponding communicator
  - CALL xios_context_initialize("context_id", comm)
  - All processes of the communicator must participate to the call
- Servers has a special channel to listen context registration
- After received a registration request, intercommunicator between client code and servers are created, and so MPI message can be routed.

Communicator & context registration

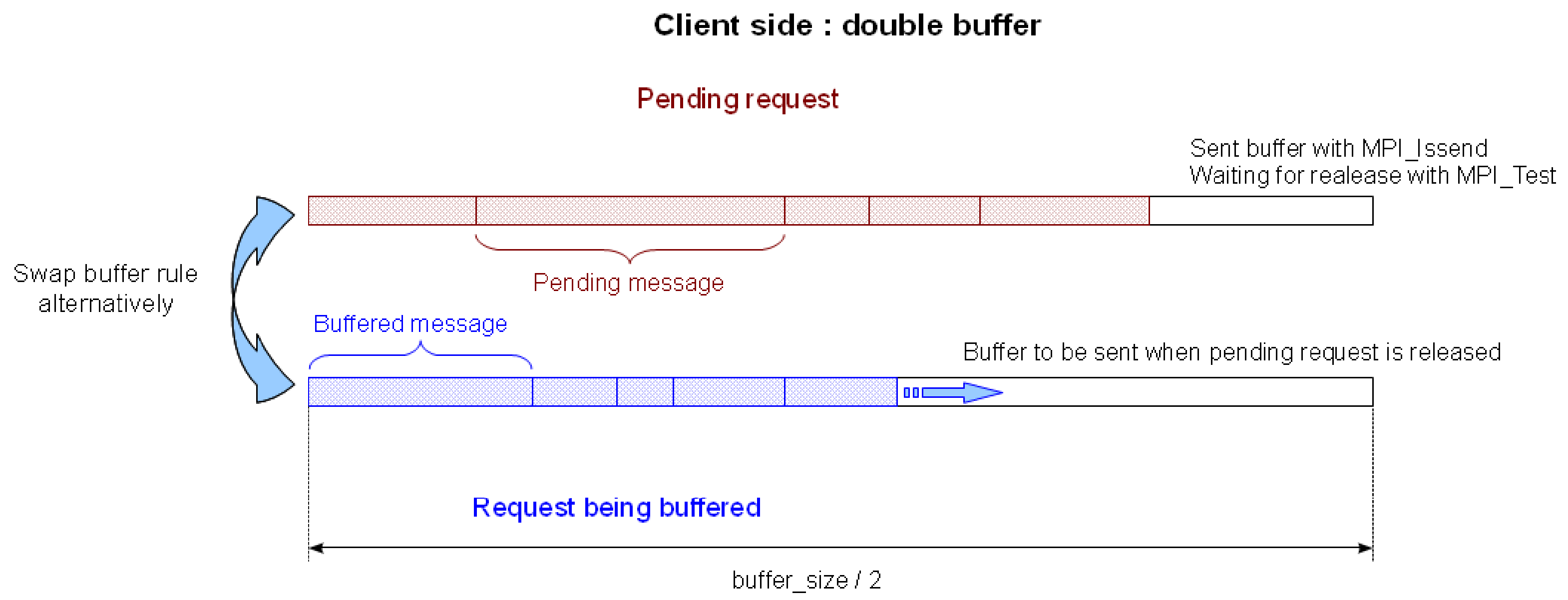
- Each context has its own unique intercommunicator with the servers.
  - None interference is possible between different context request.
- Context registration may be done at any time.
- More than one context registration my be done inside a code, with same or different communicator
- Servers can manage context registration from different client codes.

Communication protocol

- Communication between client and server use principle of RPC (Remote Proceduring Call) programming (like CORBA) through MPI.
  - A message is self-descriptive and contains information from provenance, for routing to destination and data.
  - A message is filled from client side by packing data arguments
  - When the message is received at server side, it is partially analyzed and routed to the targeted class method.
  - The message is unpacked by the same way it was packed and the corresponding method is called with the unpacked arguments.

- Zoology
  - **Message** : concatenation in a buffer of different calling arguments.
  - **Request** : concatenation of several messages. It will be sent/received through MPI layer by asynchronous call.
  - **Event** : set of several message from different client, but targeted to the same server method.
    - As messages can be received in disorder, messages from a same event are identified by the same unique timeline Id (integer).
    - After reception, events are processed in order of timeline Id.

Client Side protocol



- When adding a new message, check if the pending request can be release.
  - use asynchronous MPI_TEST
- if yes, then sent the active buffer and swap buffer rule.
  - use asynchronous MPI_ISSEND
  - Released buffer becomes active buffer.
- Add new message in the active buffer.
- if the active buffer is full, the loop on the pending buffer until it will be released.

Server-Side Protocol

- Server side protocol
  1- Loop onto registered context

  2- Loop onto client of a context

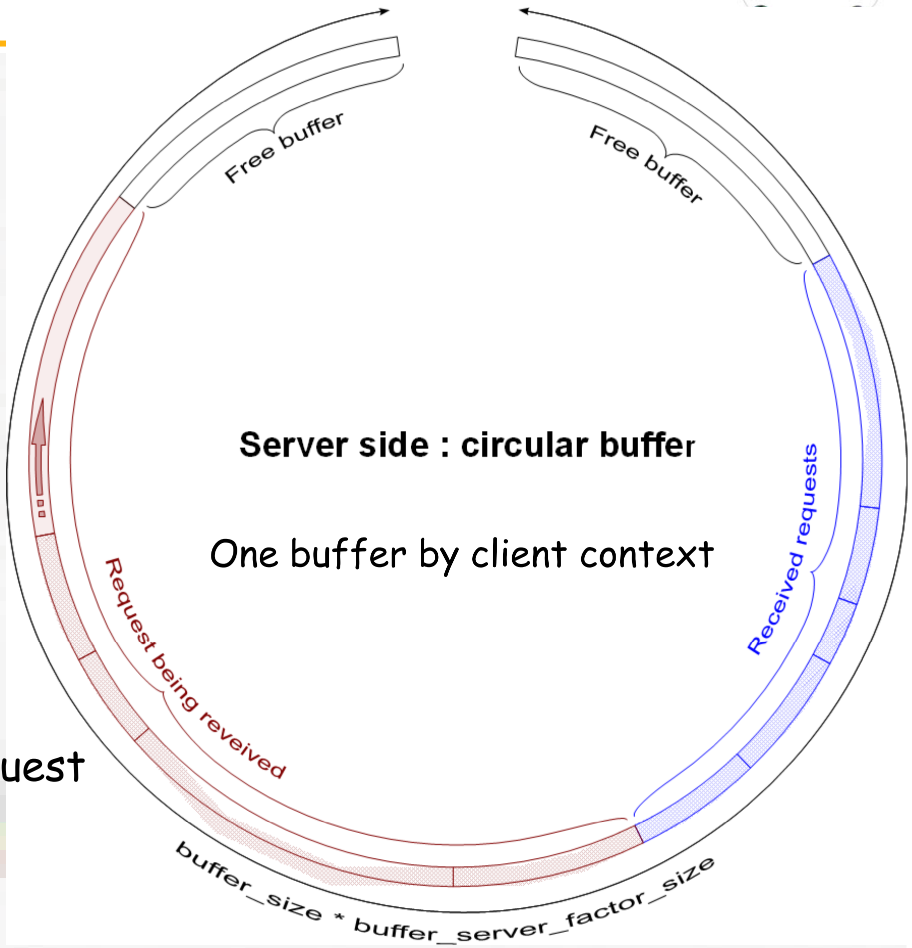
  3- If a request is being received
  - jump to next client

  3- Else check if a message is available
  - using asynchronous MPI_IPROBE
  - if not jump to next client

  3- If yes, check the buffer free size
  - if buffer is full jump to next client

  3- If enough space in buffer, receive request
  - use asynchronous MPI_IRECV

  3- Next client…



  2- If no request is available for all client, then process received event
  - Check if the next event identified by timeline Id is completed
  - route the event to the targeted method
  - release corresponding buffer

IO Layer

IO layer

- IO layer is very modular, and so new IO layer can be easily added
- For moment, only netcdf4 IO layer with HDF5 has been implemented.
- Two mode are possible : multiple_file or one_file
  - mode can be set up by file with the file attribute : type = "multiple_file"/"one_file"
- Multiple file : one file by server
  - No parallel access is used
  - File is suffixed by server rank
  - building phase is needed
- One_file : a single complete file is written
  - use parallel collective or independent access
- For moment, no investigation has been done to manage chunk and compression

Performance

- Nemo test-case with realistic configuration

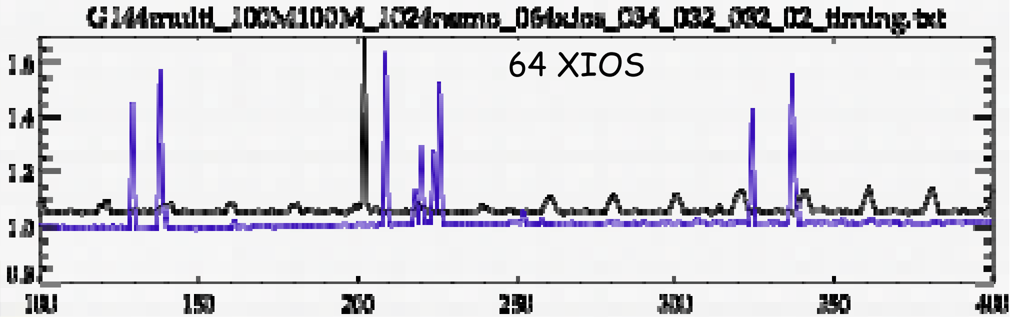
Configuration: GYRE 144 (4322×2882)

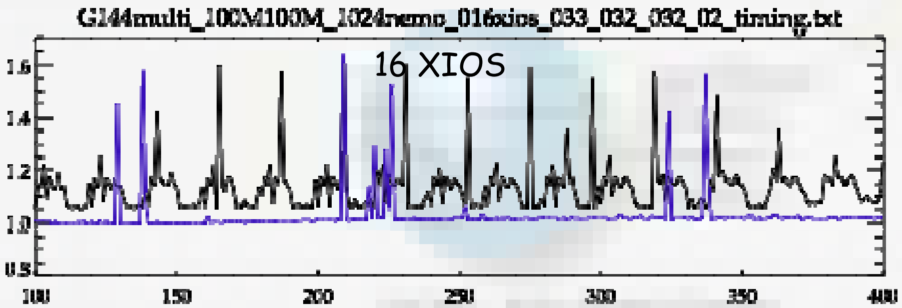
rdt=180, run 720 time-steps (36 hours)

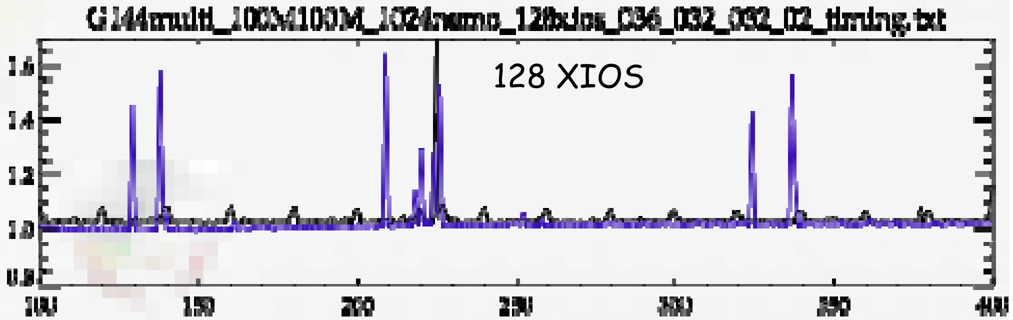
Hourly output – no output

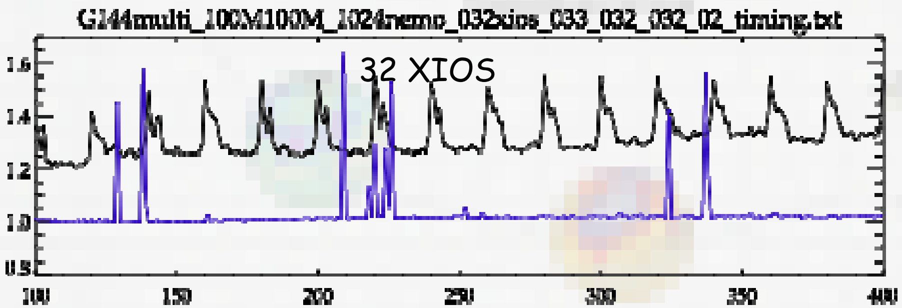
246 Go written into 4 files

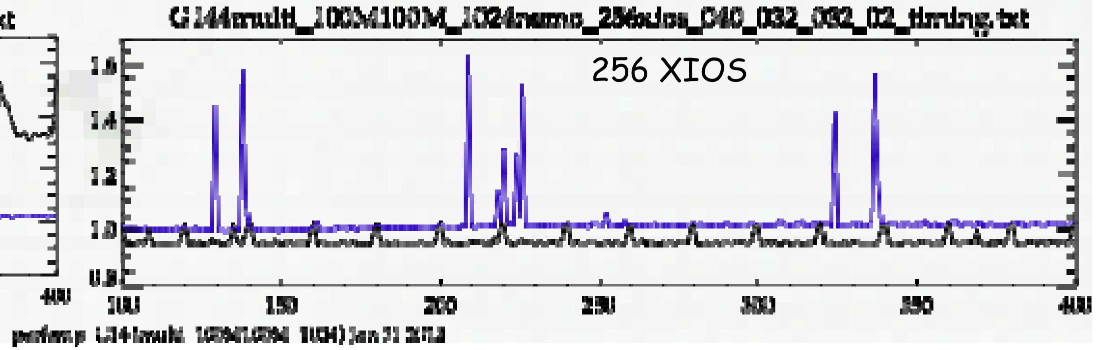
```
62G Feb 25 04:36 BIG1h_st32_1h_grid_T.nc
28G Feb 25 04:37 BIG1h_st32_1h_grid_U.nc
26G Feb 25 04:31 BIG1h_st32_1h_grid_V.nc
130G Feb 25 04:35 BIG1h_st32_1h_grid_W.nc
```







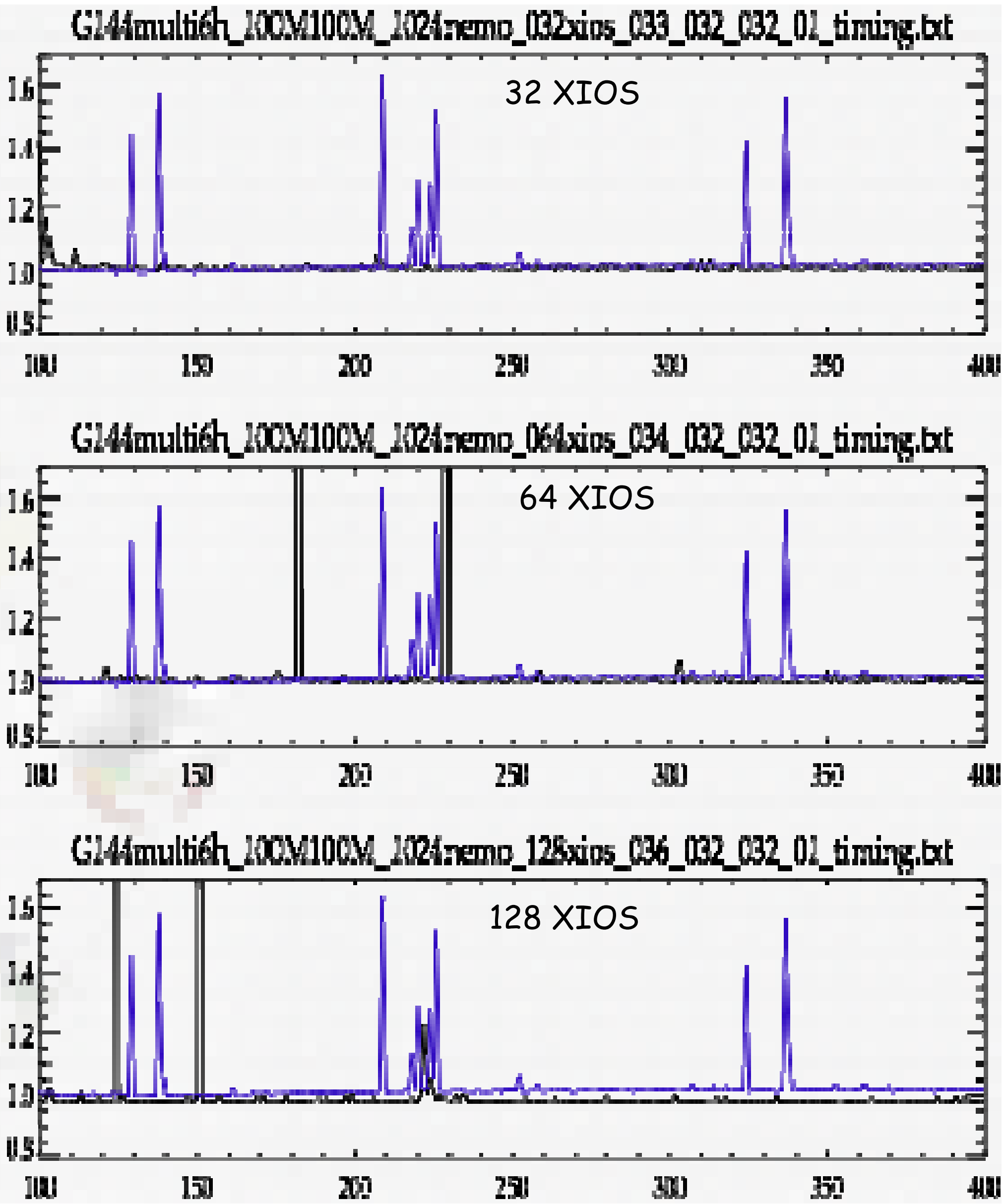


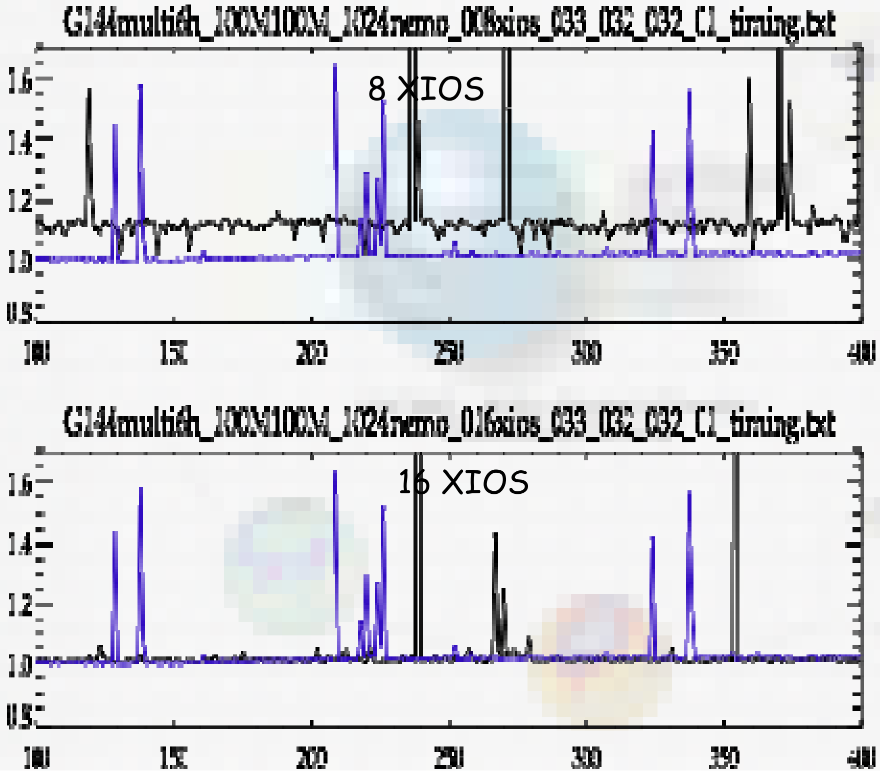


Performance

- lighter test case

6 Hours output – **no output**





Performance

- Good results with multi_file option

- But bad performance with one_file option

- Very fresh result (yesterday), none investigation has even been achieved to elucidate the problem.
  - Chunking problem ?
  - Collective parallel access Vs independent parallel access ?
  - Lustre interference ?
  - Request size too short ?

- Will be investigated in priority during the next weeks

Conclusion and perspectives

- XIOS library begin to be mature
- First result on performance are encouraging, some improvement to do on the parallel IO.
- People are very happy with the flexibility of the IO management.

Perspectives

- Short term
  - Parallel IO improvement
  - Documentation up to date
    - User documentation
    - Reference documentation using Doxygen
  - Spatial operation between field
  - Integrate XIOS into the whole IPSL coupled model : NEMO, LMDZ, ORCHIDEE, INCA + OASIS.
  - Using trac for bug and request management.
  - Other collaboration ?

Conclusion and perspectives

- More longer term
  - Management of restarts.
  - Possibility to manage asynchronous reading, ie for forcing field.
  - Extend functionality of XIOS on unstructured grid (ICOMEX)
  - Managing grid transformation like interpolation and resolution upscaling (ICOMEX).
  - Managing global operation like global mean or zonal mean.
  - And many more idea…