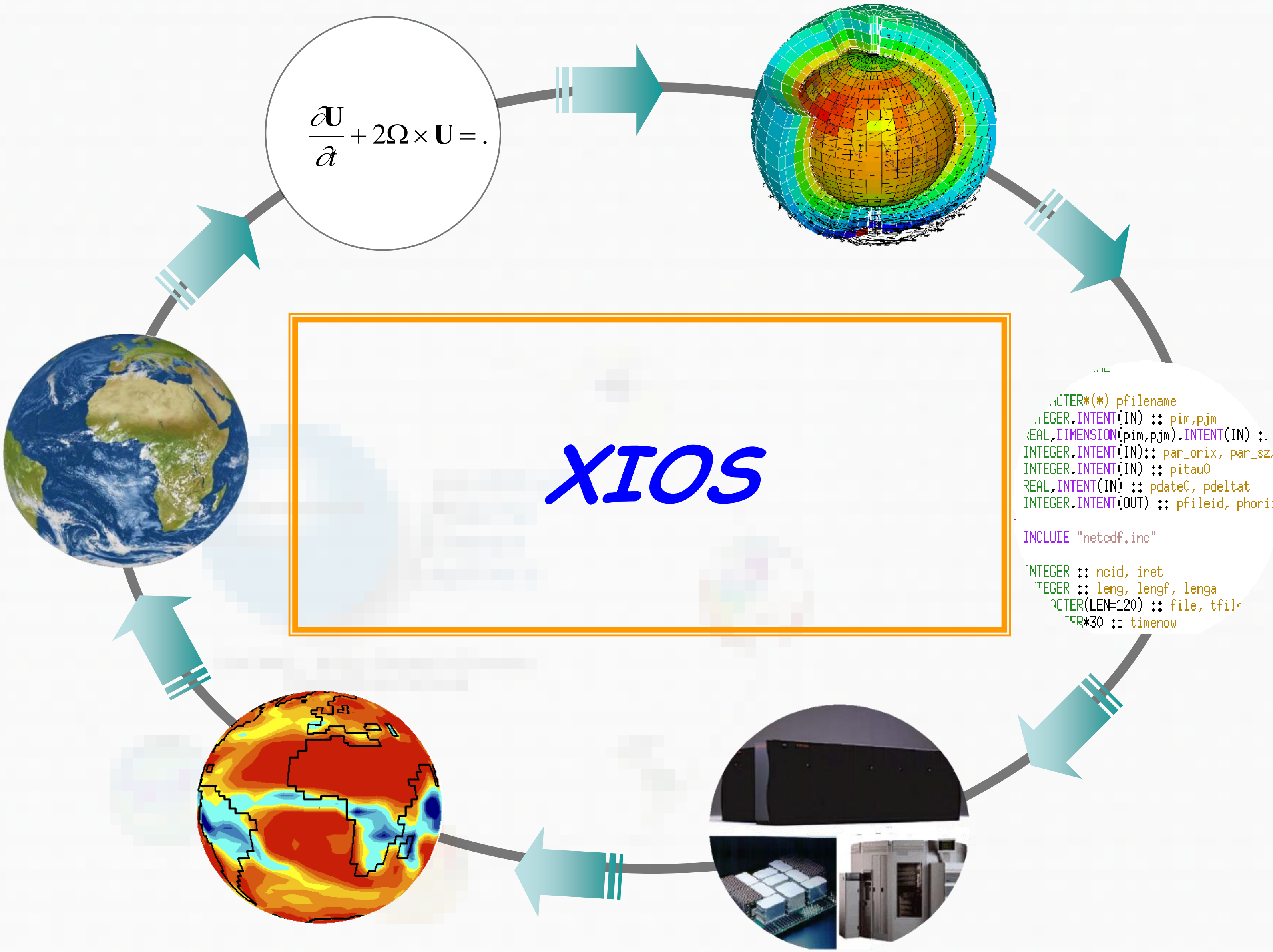
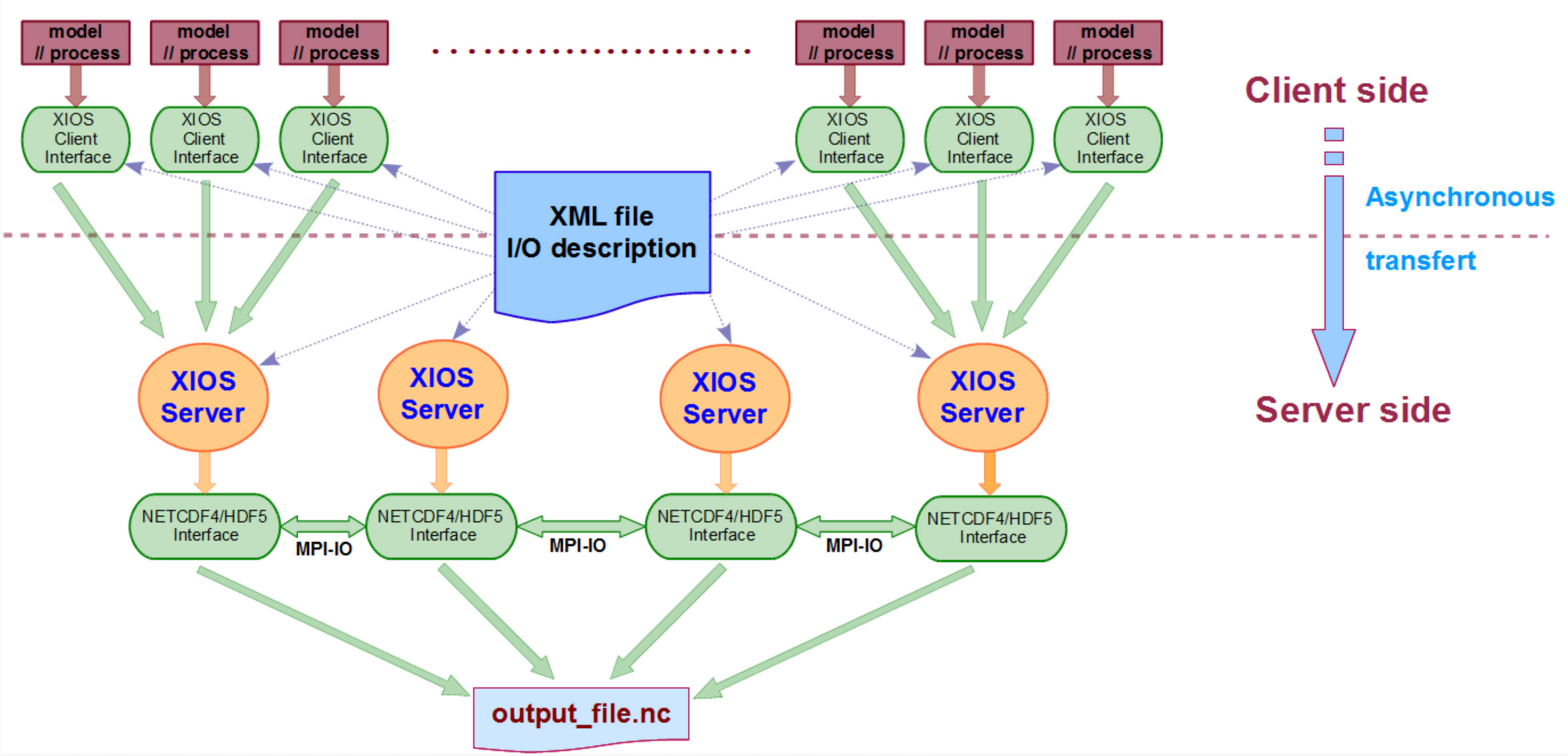
Yann Meurdesoif(LSCE-IPSL), H. Ozdoba, A. Caubel, O. Marti

# XIOS



## 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.



# XIOS - Motivation

## 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
    - Targeted for large core simulation (> ~10 000) on climate coupled model.
    - 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

- 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.
- Mid 2012 : XIOS integration into NEMO and testing.
- Now : ~ 35000 code lines under SVN : http://forge.ipsl.jussieu.fr/ioserver/browser/XIOS/trunk

## 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++), IBM (xlf/xlc) and Cray compiler

# Simple "Hello World" XML file

- Output averaging field: field_A in the one day frequency file : hello_world.nc

```
<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>

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

  </context>
</simulation>
```

# XIOS - Fortran Interface

- Interoperability C/C++/Fortran through Fortran 2003 normalization feature.
- Every element in XML tree file can be created or fill in from code model through 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)
```

# "Hello World" : model 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")
  ! domain definition
  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()

  ! 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
```

# XML structuration

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

# XML structuration

- 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
  - time step : ts

- 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

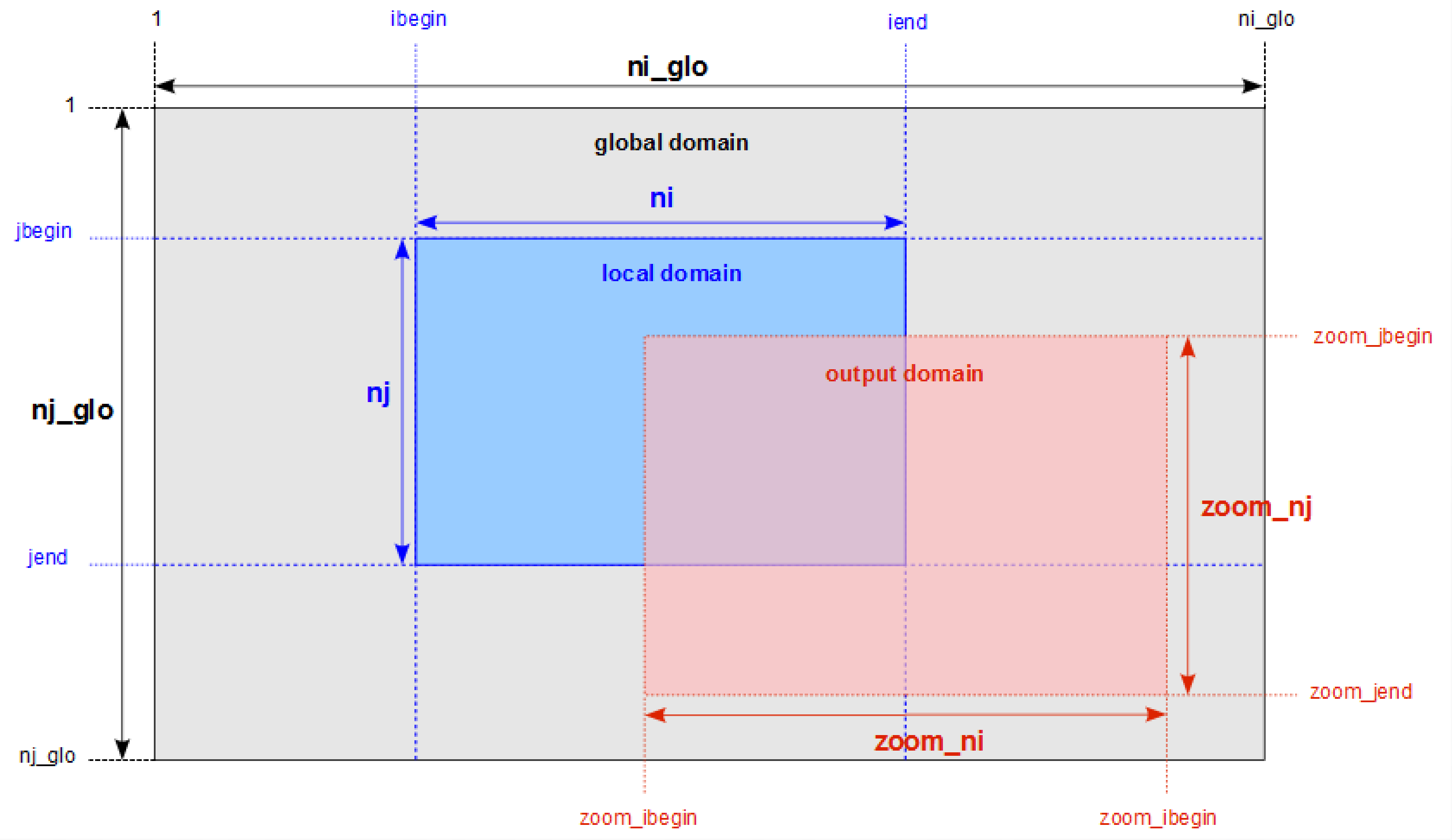
- Grid : `<grid />`
  - Only Cartesian or curvilinear grid can be manage today by XIOS
  - A grid is defining by association (referencing) of an horizontal domain and optionally a vertical axis (3D grid otherwise 2D horizontal grid)
    - `<grid id="grid_A" domain_ref="domain_A"  axis_ref="axis_A" />`

- **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]**



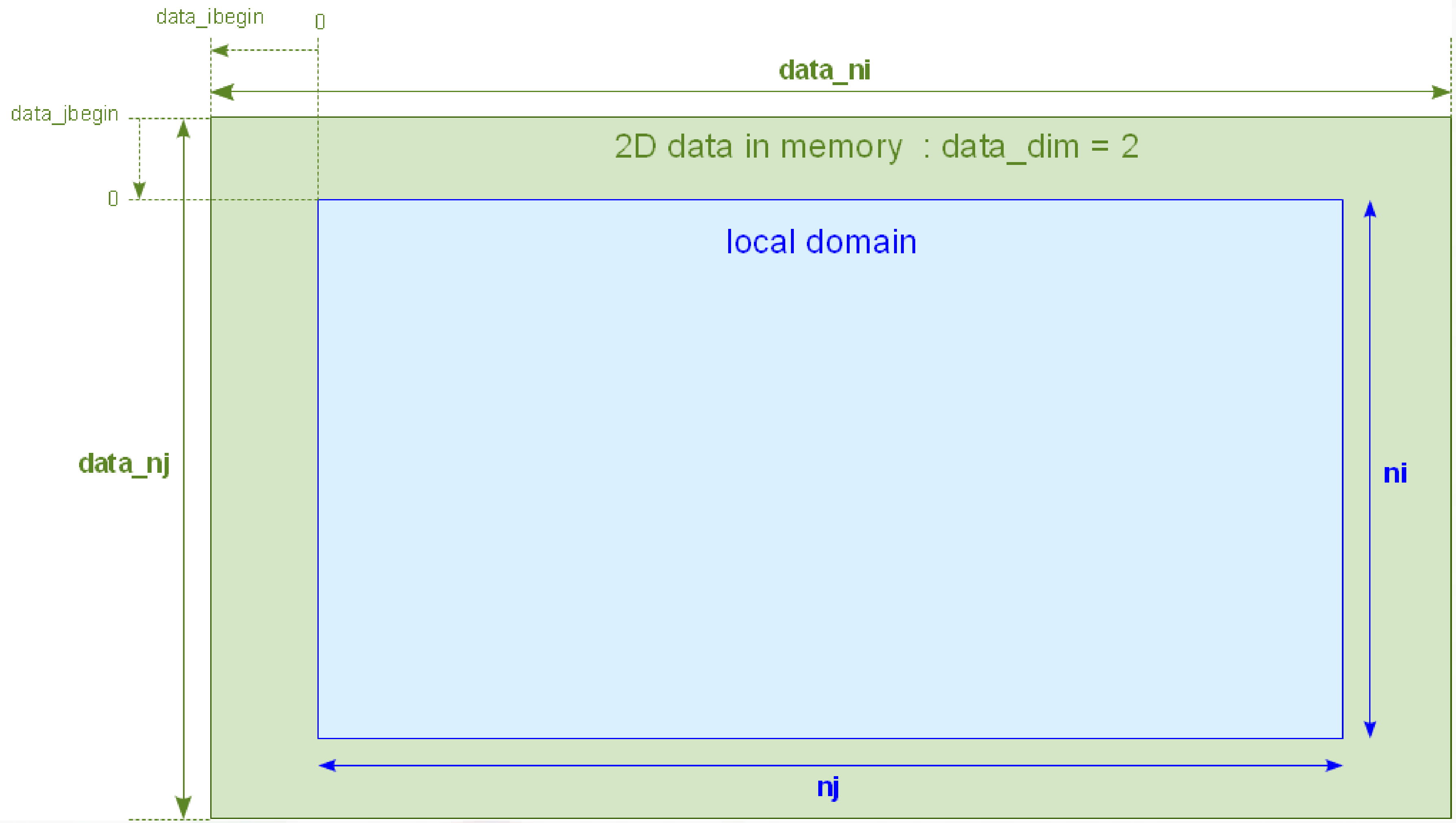
# Grids definition

- 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**)

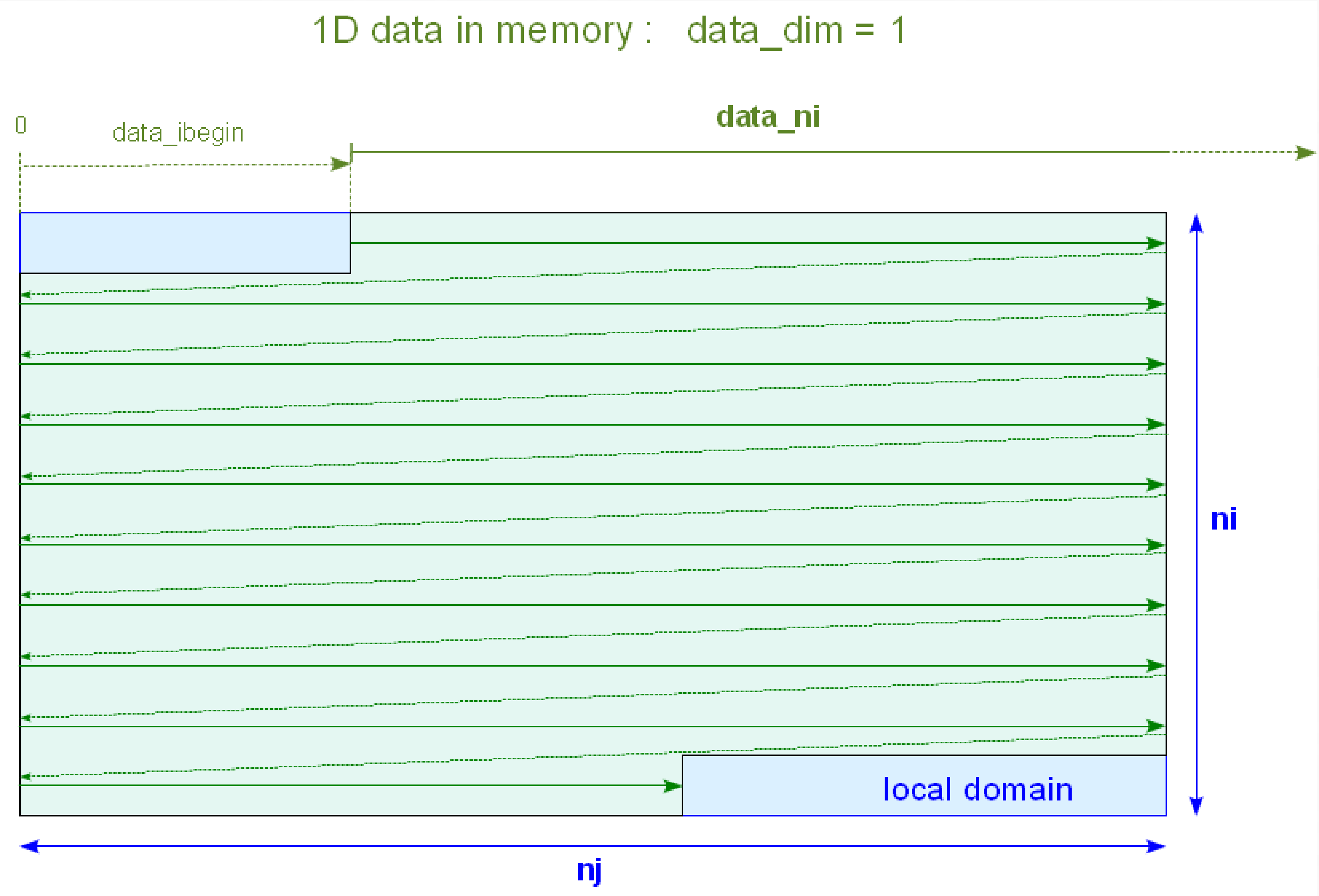
- By this way overlapping (ghost) cell can be take into account using negative offset.
  - XIOS will extract useful data from the array address.
  - default value are no overlapping cell
  - **data_ibegin=0**, **data_jbegin=0**, **data_ni=ni**, **data_nj=nj** : mapped to local domain

- 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



# Fields

## 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)

# Fields

- **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 variable
  - Fields can be combined together to create new field which can be output

```
<field id="A" />
<field id="B" />
<field id="C" > (A + B) / (A*B)   </field>
<field id="D"> (this + exp(C))/3. </field>
```

  - Operation are performed on all grid point.
  - involved field must be defined one the same grid.

- Can be mixed with temporal operation : @ operator
  - ie : output the monthly average of the daily maximum of the temperature

```
<field id="Temp" operation ="maximum" unit="K"/>
< file name ="output" output_freq ="1mo">
  <field name="T" operation ="average" freq_op="1d" > @Temp </field>
</ file >
```

- Other main attributes :
  - **name, long_name, unit, enabled, level, prec, default_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
    - **output_level** : fix the output level (related to "level" field field attribute)
    - **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
- Data transfert Client->Server are totally asynchronous :
  - Using non-blocking request.
  - 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

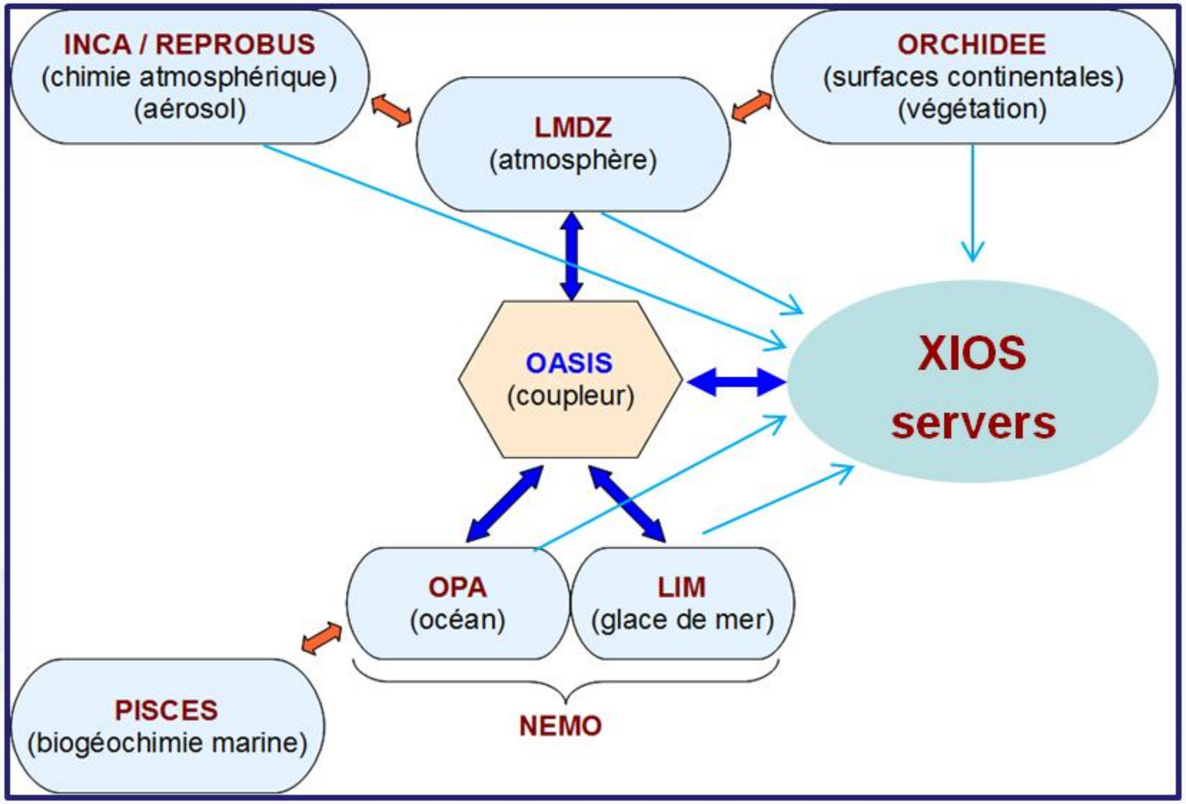
- In online (attached) mode, client codes are linked with the XIOS library and perform themselves writing on disk.
    - May suffer of computation time penalty during writing.

- 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
```

## Interaction with a coupled model

- XIOS has been designed to work within a coupled model
- A same pool of servers can manage several model output
- XIOS is also interfaced with the OASIS coupler
- switch parameter : using_oasis = true



# 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

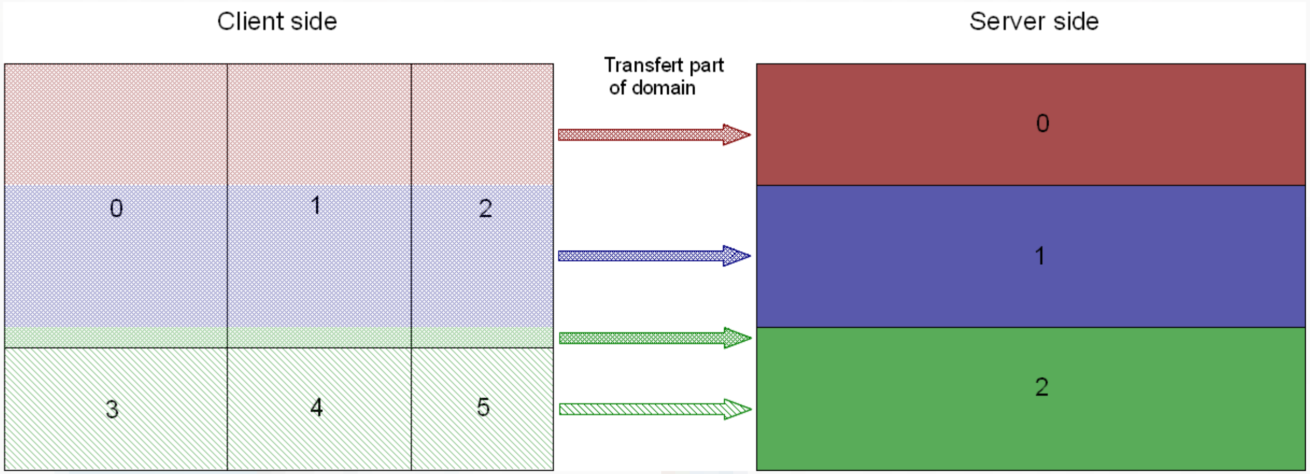
- A context is associated to a communicator.
- Before using a context, it must be registered, with its "id" and the corresponding communicator
  - CALL xios_context_initialize("context_id", comm)
  - All processes of the communicator must participate to the call
- Servers has a special channel to listen 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.

# Data distribution between clients and servers

- Data distribution on client is defined by the model
- Data distribution on server is equally distributed over the second dimension
  - Client can communicate with several servers.
  - Server can receive data from several clients.




- Field Data are sent to servers only at writing time
- Use only MPI point to point asynchronous communication
  - MPI_Issend, MPI_Irecv, MPI_Test, MPI_Probe...
  - No synchronization point between clients and server, and between servers
  - No latency cost, communications are overlapped by computation
  - Writing is also overlapped by computation

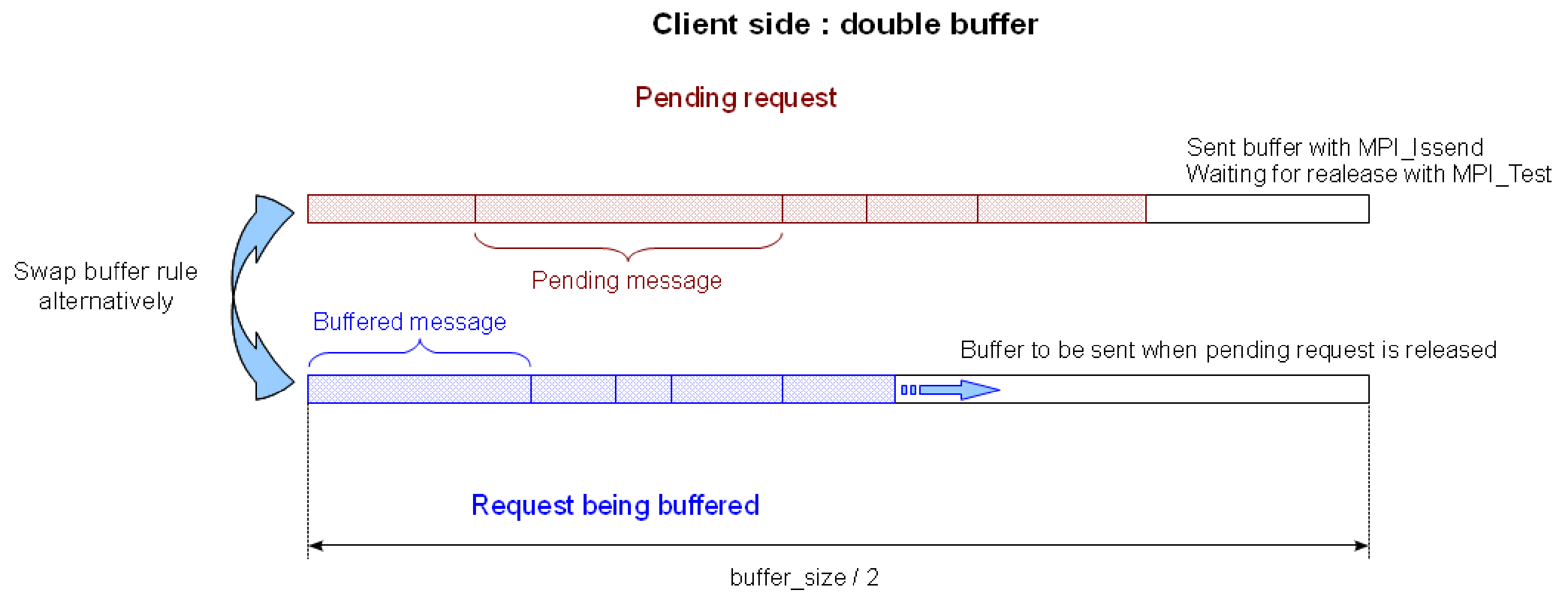
# Communication protocol

- 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

## 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

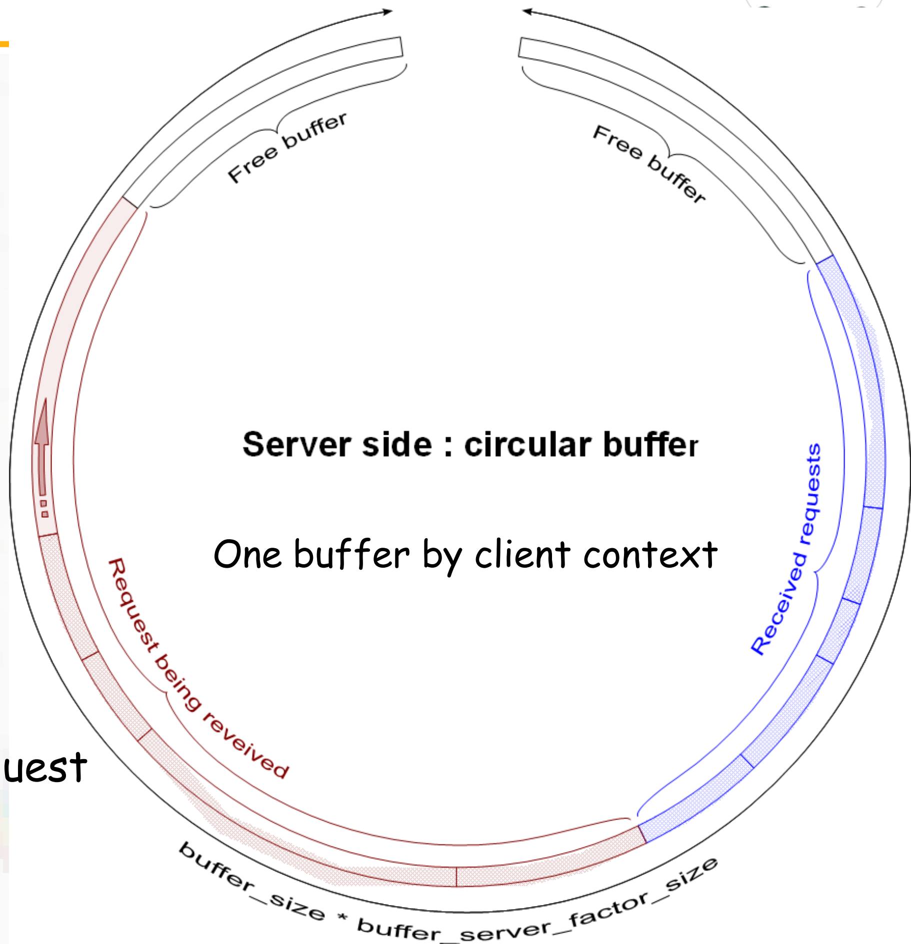
3- If a request is being received
- jump to next client

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...



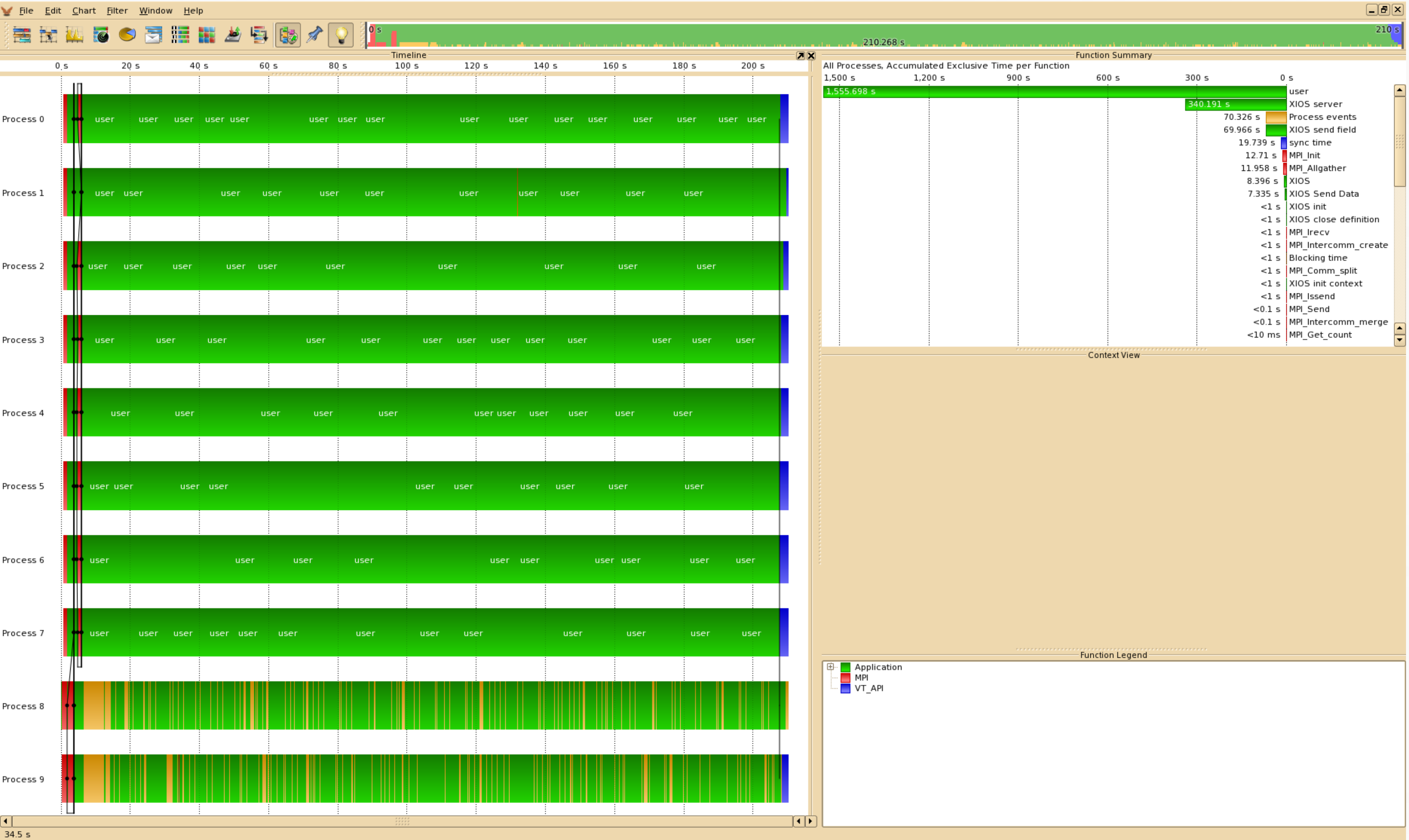

2- If no request is available for all client, then process received event
- Check if the next event identified by timeline Id is completed
- route the event to the targeted method
- release corresponding buffer

# Load balancing – coupled model

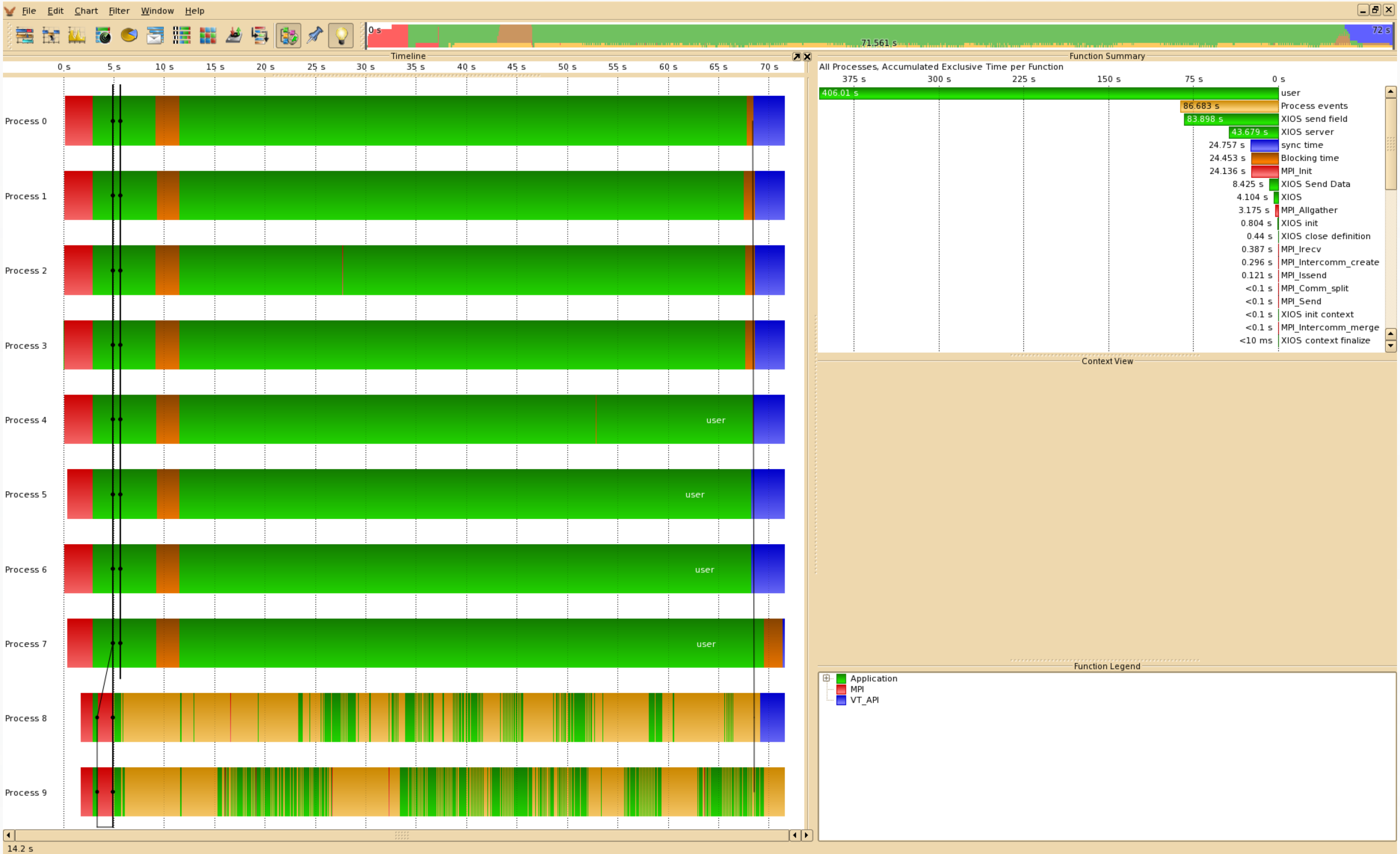
## Load balancing…

- Data flux from clients >> server writing rate capability
  - When buffers are full, XIOS switch in blocking mode
  - Wait until some buffer parts are released => impact on performance
- To avoid this, add more servers
  - a diagnostic is done at the end of the job.

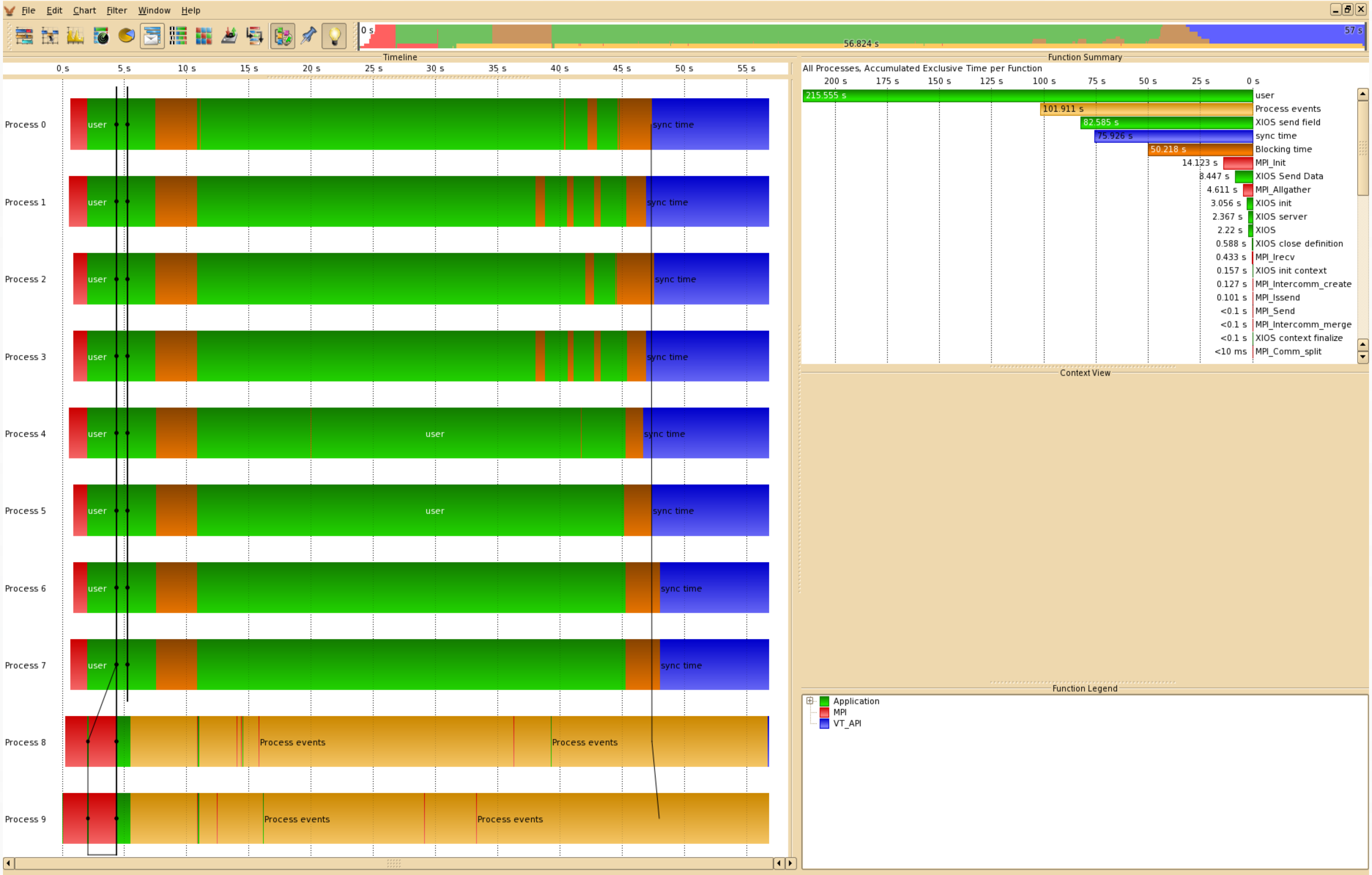
# 8 clients – 2 serveurs : temps par itération 80 ms



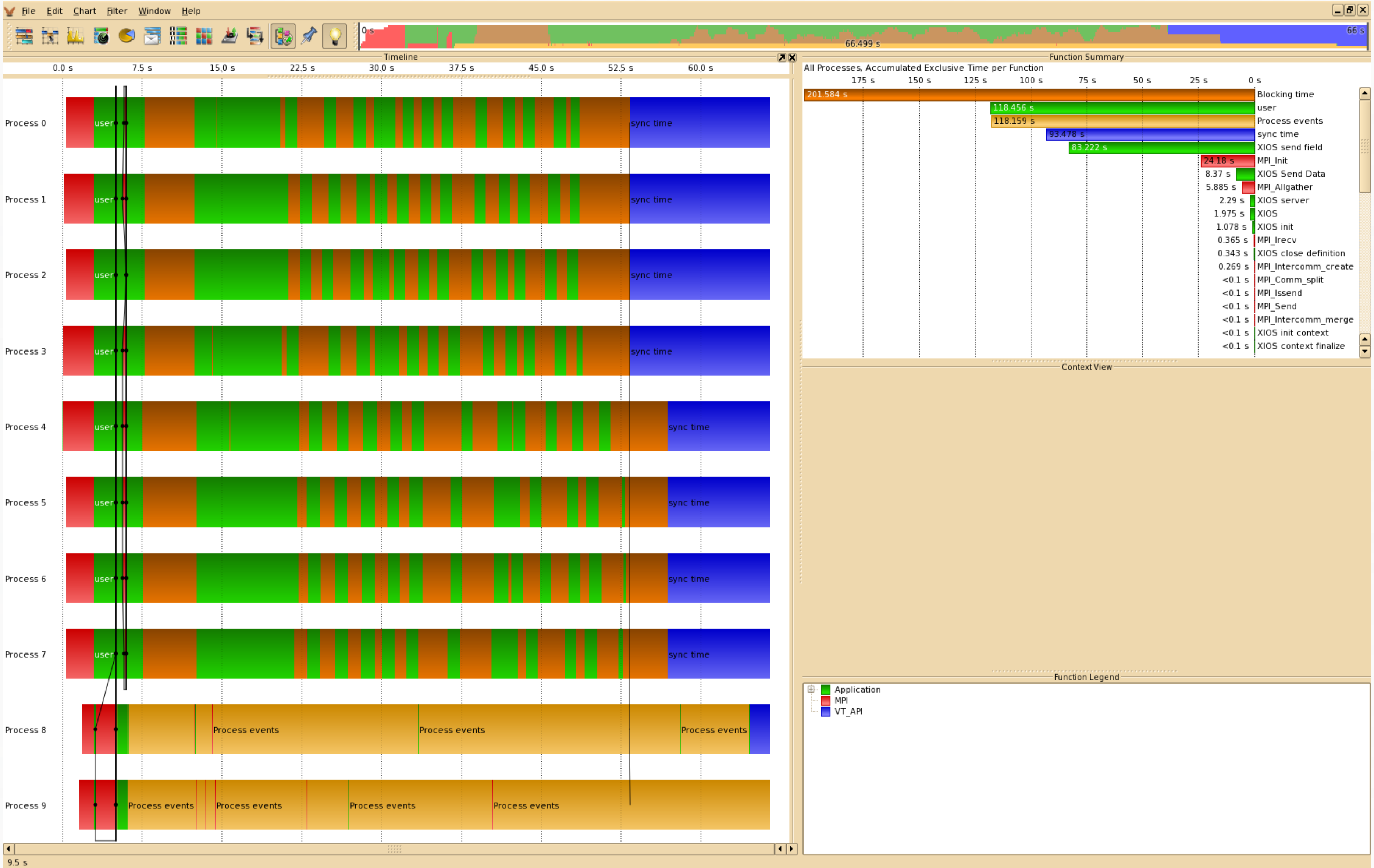
# 8 clients – 2 serveurs : temps par itération 20 ms



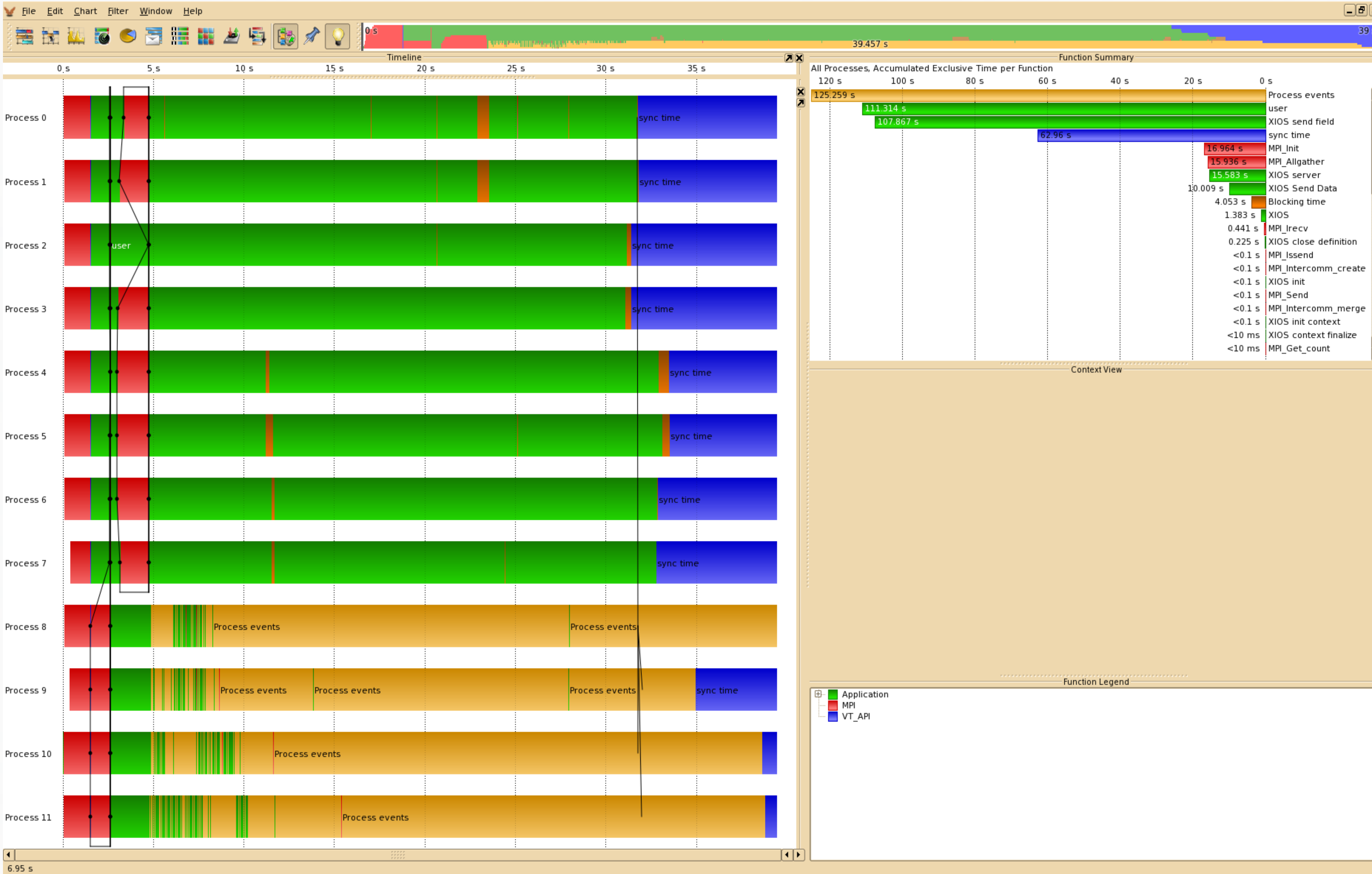
# 8 clients – 2 serveurs : temps par itération 10 ms



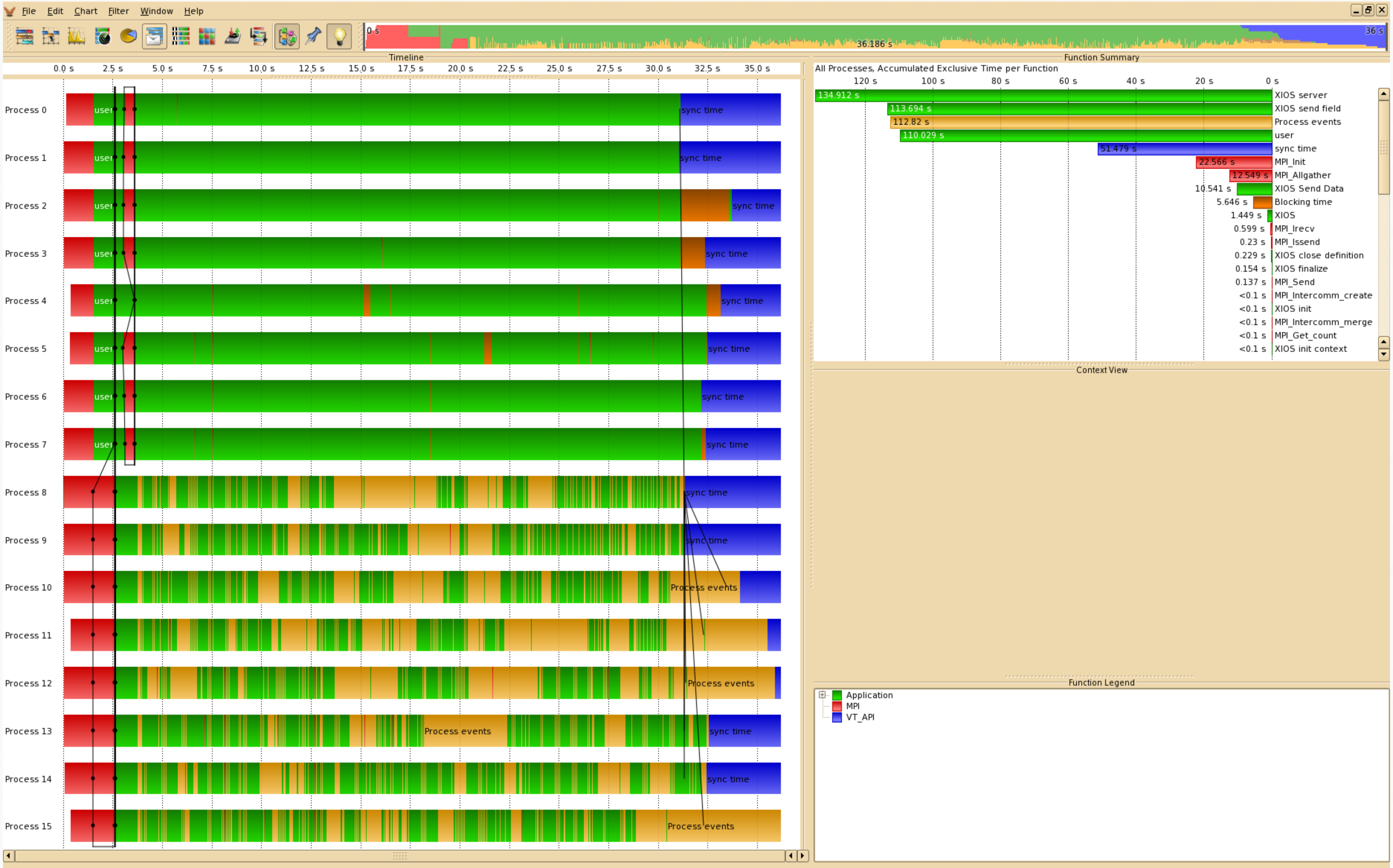
# 8 clients – 2 serveurs : temps par itération 5 ms



# 8 clients – 4 serveurs : temps par itération 5 ms



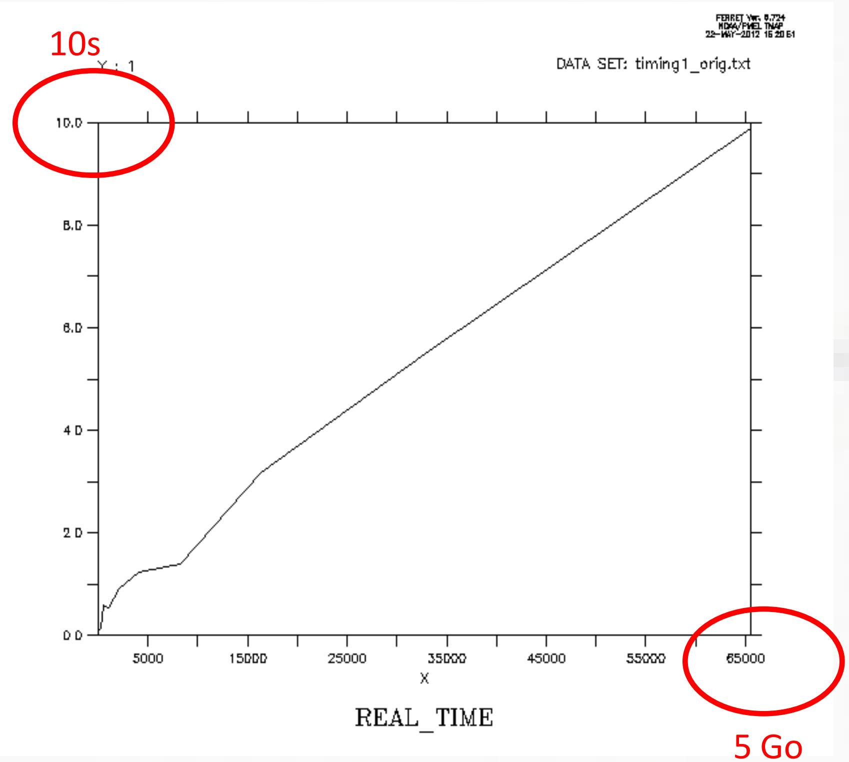
# 8 clients – 8 serveurs : temps par itération 5 ms

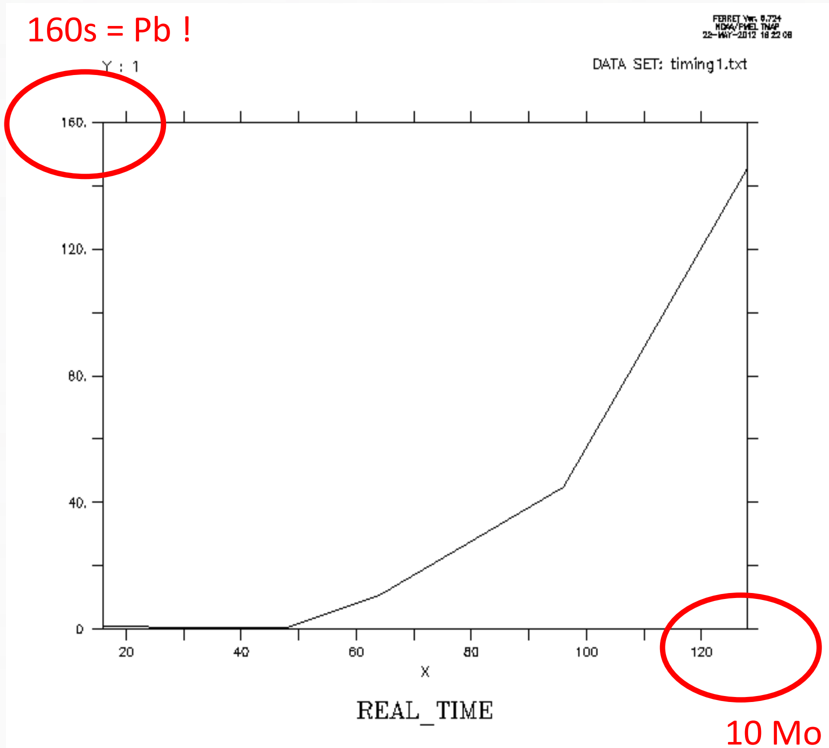


# IO Layer

- For now, output layer use only NETCDF4/HDF5 parallel library.
  - optionally netcdf3 can be used, but no parallel support
- 2 modes are possible : "one_file" or "multiple_file"
  - Fixed using file attribute : type="one_file"/multiple_file"
- "Multiple_file" mode
  - One file is output by each XIOS servers
  - rebuilding phase is needed at post-treatment to obtain a global file
- "One_file" mode
  - All XIOS servers write simultaneously in a single file => no rebuilding phase
  - Use MPI/IO layer to aggregate file system bandwidth
- But achieve good performance with netcdf4/hdf5/MPI_IO layer is very challenging
  - strongly file system dependent
  - a lot of recipes to avoid very bad performance
  - a lot of work done for improving performances.

# Encountered IO problems on CURIE




– *Multiple file on 16 CPUs : 1 file by process = 16 files*

– *Single file on 16 CPUs : 1 rebuilt file (collective access or independent access)*

# NEMO test case on Curie tier0 computer (S. Masson)

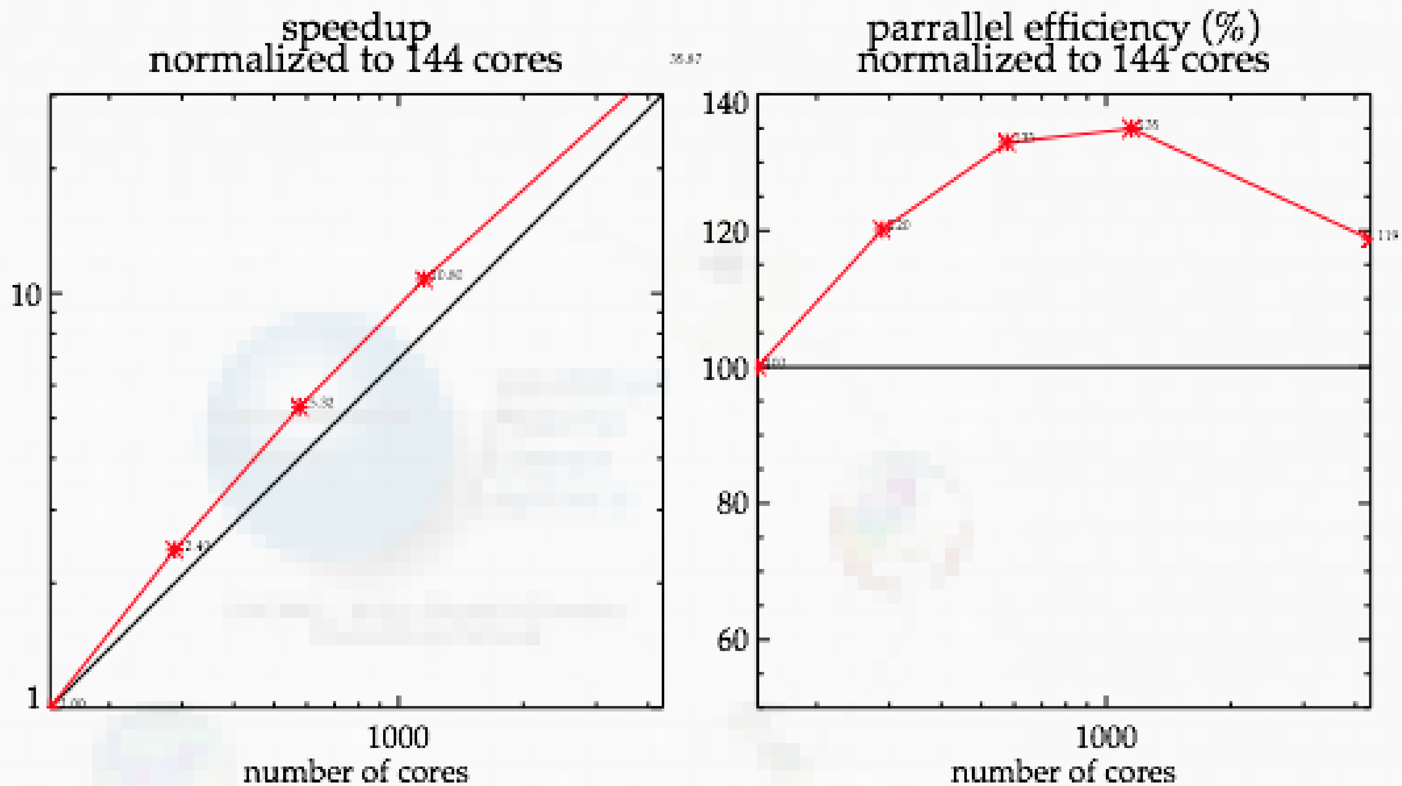
- Very huge NEMO configuration : 1/12th degrees global
  - GYRE 144 : 4322x2882x31 , up to 8160 cores
  - Run on Curie petaflopic Bull computer : 1.6 PFlops, Intel Sandy-Bridge core
  - 6 days simulation (2880 time steps), hourly means output : 300s run, ~1.1 Tb
  - **3.6 Gb/s, 13 Tb/hour, 312 Tb/day, 9.4 Pb/month (real time)**

- File system : Lustre 150 Gb/s (global) theoretically
  - In practice with an optimal MPI/IO simple parallel write test-case in a single file
    - must tune the number of OST used.
    - peak ~ 20 Gb/s, average 10 Gb/s.
  - With NETCDF4/HDF5/MPI_IO layer on an ideal test case
    - only MPI_IO call : ~ 8 Gb/s
    - whole < 5 Gb/s average

# Multiple file mode

- Works fine, good scaling up to 8160 NEMO cores with 128 XIOS servers

CURIE Fat Nodes: NEMO 3_4_b GYRE Big IO multi_file, jp_cfg = 144




| | | | | | |
|---|---|---|---|---|---|
| ncores: | 144 | 288 | 576 | 1152 | 4352 |
| timing: | 8000 | 3330 | 1505 | 741 | 223 |

# One file mode

- More challenging, recent results...
  - Gyre 144, daily mean output
    - 8160 NEMO, 32 XIOS : works perfectly
      - **+1.5% for IO** < computer jittering
  - Gyre 144, 6 hours mean output
    - 8160 NEMO, 32 XIOS
      - **+5% for IO**
  - Gyre 144, hourly mean output
    - 8160 NEMO, 128 XIOS
      - Extreme testcase, close to NEMO strong scalability limit.
      - Close to filesystem capability bandwith => we obtain ~ 3.6 Gb/s
      - **+ 15-20% for IO**

    - Do we need to improved this result ?

    - Are we able to store all this amount of data ?

# Conclusions et perspectives

- XIOS fonctionne plutôt bien
  - Amélioration des performance des écritures HDF5 parallèles.
  - Test de parallel-netcdf ?

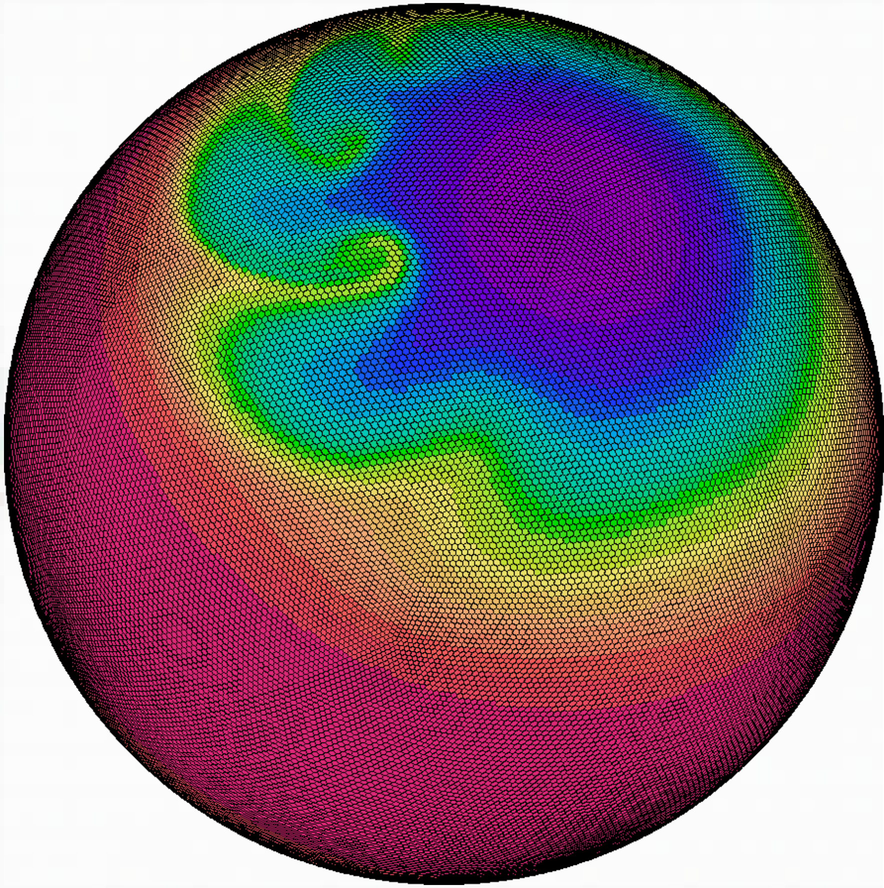
- XIOS : intégration dans l'ensemble des composantes du modèle couplé de l'IPSL.
  - NEMO : officiellement adopté par le consortium NEMO (France, UK, Italie)
    - Intégré dans la prochaine release, début 2013.
  - ORCHIDEE : en cours..
  - INCA
  - LMDZ

- Collaboration extérieure
  - LGGE (Grenoble) modèle régional MAR (H. Gallée) : en cours
  - Météo-France/CNRM (S. Sénesi)

- Collaboration Européenne ?
  - L'IPSL a-t-elle les moyens de fournir du support au-delà de sa communauté ?

# Conclusions et perspectives

- Traitements des entrées
  - Lectures des fichiers en avance de phase et envoie asynchrone
    - Gestion des forçages.
    - Gestion des restarts.
- Gestion des grilles non-structurée
  - Projet G8 ICOMEX
    - gestion des grilles icosaédrique.
  - Grilles gaussiennes (CNRM)
- Opération de regrillage
  - Opération de réduction
    - moyennes globales
    - moyennes zonales
  - Dégradation de résolution
  - Interpolation et projections vers d'autres grilles à la volée
    - E. Kritsikis (G8-ICOMEX)



## Interpolation (E. Kritsikis )

- Développement de méthode d'interpolation conservative sur la sphère
  - Ordre 2
  - Conservation exact de la masse sur les champs scalaires.
  - Grille géodésique ⇔ grille géodésique
  - Grille lon-lat ⇔ grille lon-lat
  - Grille géodésique ⇔ grille lon-lat
- Calcul des poids en ~ n log n
  - algorithme de recherche basé sur le SSTREE