

-  2 pm to 5 pm Paris time
-  4 sessions of XIOS and 1 session of dr2xml
-  Position your avatar on the orange spot
-  Turn off your microphone
-  [training Q&A shared document](#)
-  Groups of 2 or 3 for hands-on exercises

Background of the XIOS project

Get started with XIOS

- Install and compile XIOS
- Use XIOS in a model
- XML syntax
- XIOS component (context, calendar, grid, axis, domain, file, etc.)

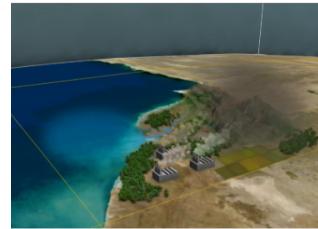
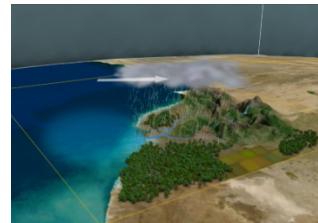
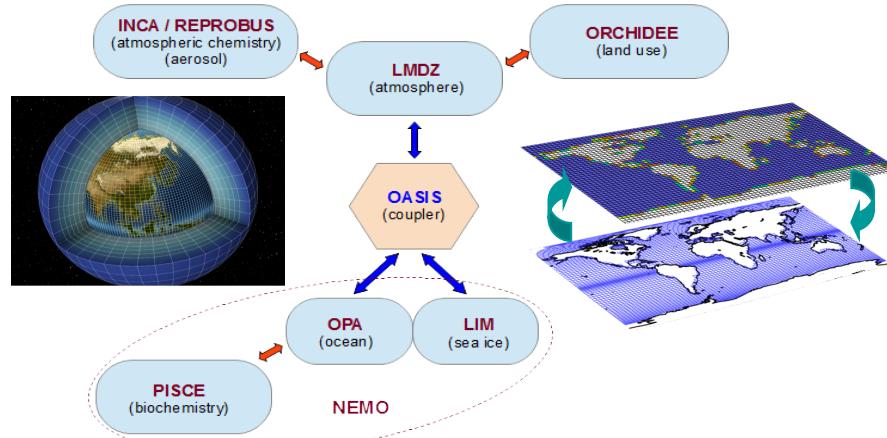
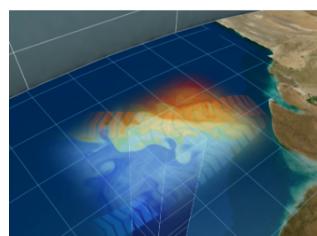
Get further with XIOS

- XIOS data distribution
- Use XIOS to read a file
- XIOS temporal filters
- How to perform data transformation in XIOS
- Activate the workflow graph in XIOS

How to improve the performance with XIOS

- Client-server mode of XIOS
- What is XIOS buffer, how it works?
- How to understand the XIOS report?
- How to parametrize XIOS?
- How to debug with XIOS?

Context : IPSL Earth System Models



Complex coupled model, long simulations, a lot of data generated...

IPSL in the past Coupled Model Inter-comparison Project phase 6 (CMIP6)

- Since March 2018
- 850 simulations (55000 model years)
- 4 PB of data (1 PB publication ready data files)
- High frequency files (3h, 6h, daily, ...)
- Lots of metadata (title, description, unit, associated axis, ...)

CMIP7 next

- CMIP3 : 24 models x 12 experiments = 39 TB (82 340 files)
- CMIP5 = 50 x CMIP3
- CMIP6 = 20~50x CMIP5

3 main challenges for climate data production

Efficient management of data and metadata definition from models

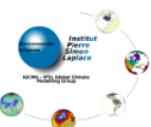
- Human cost, errors...

Efficient production of data on supercomputer parallel file system (HPC)

- 1 file by MPI process ?
 - Rebuild files (with different number of procs)
- Parallel I/O efficiency ? (not so efficient when many procs write to same file)

Complexity and efficiency of post-treatment chain to be suitable for distribution and analysis

- Files rebuild, time series, seasonal means...
- Mesh re-gridding, interpolation, compression...
- Resiliency ?



XIOS is addressing all these challenges

Efficient management of data and metadata definition from models ?

- Using an external XML file parsed at runtime
- Human readable, hierarchical

Efficient production of data on supercomputer parallel file system ?

- Dedicated Parallel and Asynchronous I/O server

Complex and efficient post-treatment ?

- Integrate internal parallel workflow and dataflow
- Managed by external XML file
- Post-treatment can be performed "in situ "

XIOS is a ~12 years old software development

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

+ XIOS : ~ 130 000 code lines, written in C++, interfaced with Fortran models

- Open Source CECILL Licence
- Code versioning : SVN (subversion)
 - ▶ XIOS 2.5 (stable) : forge.ipsl.jussieu.fr/ioserver/svn/XIOS/branches/xios-2.5
 - ▶ XIOS trunk (dev) : forge.ipsl.jussieu.fr/ioserver/svn/XIOS/trunk

+ Used by an increasing variety of models

- IPSL models : NEMO, LMDZ, ORCHIDEE, INCA, DYNAMICO
- IGE (MAR), Ifremer (ROMS, MARS3D)
- European NEMO consortium
- MétéoFrance / CNRM : Gelato, Surfec, Arpège climat (CMIP6 production)
- European models : MetOffice (HadGEM, MONC, GungHo), ECMWF (Open IFS, EC-EARTH)

Web site : wiki page

- <http://forge.ipsl.jussieu.fr/ioserver/wiki>
- Ticket system management and sources browsing : TRAC
- Documentation : on wiki page and under SVN (doc/ directory)
 - ➡ Reference guide : [xios_reference_guide.pdf](#)
 - ➡ User guide : [xios_user_guide.pdf](#)
- Support mailing list : subscribe yourself
 - ➡ XIOS users list (users support) : xios-users@forge.ipsl.jussieu.fr
 - ➡ XIOS developers list : xios-dev@forge.ipsl.jussieu.fr
 - ➡ XIOS team (non public) : xios-team@forge.ipsl.jussieu.fr

XIOS Team

- Yann Meurdesoif (CEA/LSCE - IPSL)
- Arnaud Caubel (CEA/LSCE - IPSL)
- Yushan Wang (LSCE)
- Julien Derouillat (CEA/LSCE - IPSL)
- Olga Abramkina (IDRIS)

Download XIOS

 `svn co http://forge.ipsl.jussieu.fr/ioserver/svn/XIOS/trunk`

Compile XIOS

 `./make_xios`

Hands-on 0

Option	Value	Default	Description
<code>-- arch</code>	<code>arch_name</code>		Mandatory. Define target architecture
<code>-- avail</code>			Show available target architectures
<code>-- prod</code>			Compilation in production mode (default)
<code>-- debug</code>			Compilation in debug mode
<code>-- full</code>			Generate dependencies and recompile from scratch
<code>-- build_dir</code>	<code>build_directory</code>		Name of the build directory
<code>-- job</code>	<code>ntasks</code>	1	To use parallel compilation with <code>ntasks</code> processus
<code>-- netcdf_lib</code>	<code>netcdf_par</code> <code>netcdf_seq</code> <code>netcdf_internal</code>	<code>netcdf_par</code>	Choice of netcdf library
<code>-- help</code>			Show all available options and descriptions

+ Each time step, models expose part of their data through a minimalist interface

- Identifier (ASCII string) + address (pointer) of the data

→ Output: **CALL xios_send_field("field_id",field_out)**

→ Input: **CALL xios_recv_field("field_id",field_in)**

+ External XML File :

- Describe the incoming dataflow from models (using XML attributes)
- Describe the workflow applied to the incoming dataflow
- Describe the dataflow endpoint => output to files or returned to model

+ Simplicity and Flexibility

- XML file is parsed at runtime
 - Metadata, workflow and output definition can be modified without recompiling
- Hierarchical approach using strong inheritance concept
 - Attributes are inherited from parent to child
 - Avoiding redundant definition, simple and compact
 - Very useful when you need to describe hundred's of variables

+ Full interactivity with models through the XIOS Fortran API

- Most of the XML definitions can be completed or created from model

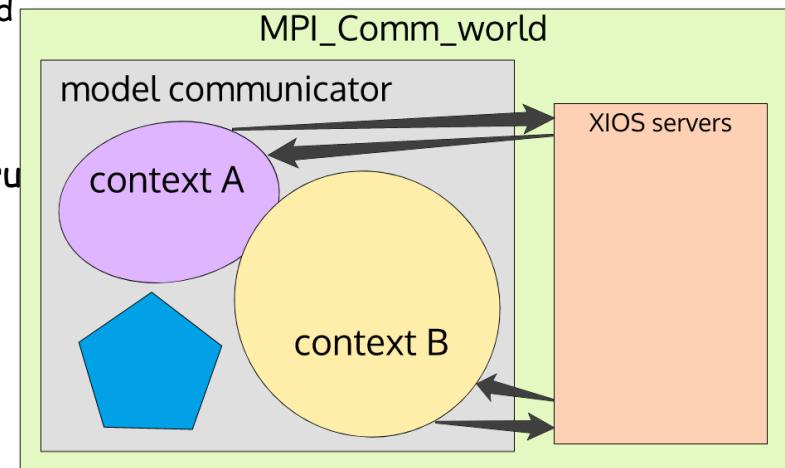
A minimal Fortran structure to be XIOS compliant

+ XIOS Initialization (mandatory)

- XML files are parsed at initialization
- **CALL xios_initialize("code_id", return_comm=communicator)**
 - ➡ "code_id" must be the same for all process rank of same model
 - ➡ XIOS split the MPI_COMM_WORLD communicator between clients and servers and return the split one for client side

+ Context initialization (mandatory)

- **CALL xios_context_initialize("context_id",communicator)**
 - ➡ "context_id" : id of the context to bind with context defined in XML file
 - ➡ **communicator** : MPI communicator associated to the context
 - ➡ Must be the same or a sub communicator of which returned at XIOS initialization
- Context initialization can be done at any time
- Different contexts can be initialized during same run
- All XIOS calls from model are collective for the associated context MPI communicator



+ Switching to a context

- **CALL set_current_context("context_id")**
 - ➡ All xios fortran calls afterwards will be related to context "context_id"

+ Complete the XML database definition

- Set missing attribute
 - ➡ Some attribute values are known only at run time
- All attribute can be set via the Fortran API
 - ➡ CALL `xios_set_element_attr("element_id",attr1=attr1_value, attr2=attr2_value,...)`
- New child element can be added
 - ➡ All XML tree can be created from Fortran interface
 - ➡ Ex : adding "temp" field element to "field_definition" group

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

+ Setting time step and other calendar specific attributes

- CALL `xios_define_calendar(type="Gregorian")` (mandatory in fortran or in xml)
- CALL `xios_set_timestep(duration)`

+ Closing context definition (mandatory)

- CALL `xios_close_context_definition()`
- Context data base is analysed and processed
- Any modification behind this point would not be taken into account and unexpected results may occur

Entering time loop and send data

- When entering a new time step, XIOS must be informed
- **CALL xios_update_timestep(ts)**
 - **ts** : timestep number
- Time step must begin to 1
- Time step 0 refers to part between context closure and first time step update
 - Only received field request can be done at time step 0
- Data can be exposed during a time step
 - **CALL xios_send_field("field_id",field)**
 - **CALL xios_recv_field("field_id",field)**
 - Sent data field would create a new flux tagged with timestamp related to the time step
 - Data can be received only if the outgoing flux have the same timestamp to the related time step

Finalize context

- All opened context must be finalized after the end of time loop
- **CALL xios_context_finalize()** close the current context.

Finalize XIOS

- After finalizing all opened context, XIOS must be finalized, servers are informed, files are properly closed and performance report is generated
- **CALL xios_finalize()**

Fortran

```
SUBROUTINE hello_world(rank,size)
  USE xios
  IMPLICIT NONE
  INTEGER :: rank, size, timestep
  TYPE(xios_duration) :: dtime
  DOUBLE PRECISION,ALLOCATABLE :: lon(:,:), lat(:,:), field (:,:)
  INTEGER :: ni, nj, ibegin, jbegin

  CALL xios_initialize("client", return_comm=comm)
  CALL xios_context_initialize("hello_world", comm)

  CALL xios_set_domain_attr("domain", ibegin=ibegin, ni=ni, jbegin=jbegin, nj=nj)
  CALL xios_set_domain_attr("domain ", lonvalue_2d=lon, latvalue_2d=lat)

  dtime%second=3600
  CALL xios_set_timestep(dtime)

  CALL xios_close_context_definition()

  DO timestep=1,96
    CALL xios_update_calendar(timestep)
    CALL xios_send_field("field", field)
  ENDDO

  CALL xios_context_finalize()
  CALL xios_finalize()
END SUBROUTINE hello_world
```

Initialise XIOS and one context

Define domain

Set time step to 1 hour

End of context definition
No more modification to the context

Enter the time loop

Free the context and quit XIOS

Hands-on 1

XML : Extensible Markup Language

- Set of rules to define a document in a format
- Both human-readable and machine-readable

Tag : a markup construct that begins with "<" and ends with ">"

- Start-tag : <.....>
- End-tag : </.....>
- empty-element tag, such as <..... />

Element : construct delimited by a start-tag and an end-tag, or consists only of an empty-element tag

- Element: <**field** > </**field**>
- Empty-element : <**field** />
- May have child elements

```
<field_group ... ...>
  <field ... ... />
  <field ... ... />
</field_group>
```

- May have content : text between start-tag and end-tag element : <**field**> content </**field**>
- Used in XIOS to define arithmetic's operations

+ Attributes : a construct consisting of a name–value pair (`name="value"`) that exists within a start-tag or an empty-element tag

- Ex : Element field has 3 attributes : `id`, `name` and `unit`
- `<field id="temp" name="temperature" unit="K" > </field>`
- `<field id="temp" name="temperature" unit="K" />`

+ Comments : begin with `<!--` and end with `-->`

- `<field> <!-- this is a comment, not a child nor a content --> </field>`
- `--` (double-hyphen) is not allowed inside comments. No nested comments

+ XML document must be well-formed

- XML document must contain only one root element
- All start-tag element must have the matching end-tag element (case sensitive) and reciprocally
- All element must be correctly nested

+ XML parser

- `rapidxml`

+ XML master file must be *iodef.xml*

- Parsed first at XIOS initialization
- Root element name is simulation
- Root element can only contain **context** type elements

+ Main element families: represent objects type stored in XIOS database

- **context** : isolate and confine models definition, no interference between them
- **calendar** : mandatory, 1-to-1 association with context
- **scalar, axis, domain**
- **grid**
- **field**
- **file** : input or output
- **variable** : define parameters for models and for XIOS parameterization

+ Each element family can be divided into 3 types (except for context)

- Simple elements : ex : <**field** />
- Group elements : ex : <**field_group** />
 - ➡ Can contain children simple element
 - ➡ Can contain children nested group of the same type
- Definition elements : ex : <**field_definition**>
 - ➡ Unique root element type
 - ➡ Act as a group element, i.e. can contains other groups or simple elements

Each element may have several attributes

- i.e. : `<file id="out" name="output" output_freq="1d" />`
- Attributes give information for the related element
- Some attributes are mandatory: error is generated if attribute not defined
- Some attributes are optional but have a default value
- Some attributes are completely optional

Attributes values are ASCII string, depending on the attribute, can represent :

- A character string : `name="temperature"`
- An integer or floating value : `output_level="3" add_offset="273.15"`
- A Boolean : true/false : `enabled="true"`
 - Fortran notation .TRUE./.FALSE. are allowed but obsolete
- A date or duration : `start_date="2000-01-01 12:00:00"`
 - See format later
- A bound array (inf,sup)[values] : `value="(0,11) [1 2 3 4 5 6 7 8 9 10 11 12]"`

+ Special attribute **id** : identifier of the element

- Make reference to the element
- Unique for one given kind of element
 - ➡ Elements with same id ⇔ same element (append, overwrite)
 - ➡ Be very careful when reusing same ids, not advised (no fixed parsing order)
 - ➡ Definition elements are equivalent to group elements with a fixed id
 - ➡ Ex: <**field_definition**> ⇔ <**field_group id="field_definition"** ...>
- **id** is optional, but no reference to the element can be done later

+ XML file can be split in different parts.

- Very useful to preserve model independency, modularity
- **id** must be the same in both xml files
- Using attribute "**src**" in context, group or definition element
 - ➡ attribute value give the name of the file to be inserted in the database

```
--- iodef.xml ---
<context id="nemo" src=".//nemo_def.xml" />

--- nemo_def.xml ---
<context id="nemo" >
  <field_definition ... >
  ...
</context>
```

Why Inheritance ?

- Attributes can be inherited from another element of same family
- Hierarchical approach, very compact
- Avoiding useless redundancy

Inheritance by grouping : parent-child inheritance concept

- All children inherit attributes from their parent
- An attribute defined in a child is not inherited from his parent
- Special attribute "id" is NEVER inherited

```
<field_definition level="1" prec="4" operation="average" enabled=".TRUE.">
  <field_group id="grid_W" domain_ref="grid_W">
    <field_group axis_ref="depthw">
      <field id="woce" long_name="vertical velocity" unit="m/s" operation="instant" />
    </field_group>
  </field_group>
</field_definition>
```



```
<field id="woce" long_name="vertical velocity" unit="m/s" axis_ref="depthw"
      domain_ref="grid_W" level="1" prec="4" operation="instant" enabled="true" />
```

Inheritance by reference

- Only for **field**, **domain**, **axis**, and **scalar** elements
 - ➡ **field_ref**
 - ➡ **domain_ref**
 - ➡ **axis_ref**
 - ➡ **scalar_ref**
- Source element inherit all attributes of referenced element
 - ➡ Attributes already defined in source element are not inherited (or is overwritten)

Don't mix up with **grid_ref** !

```
<field id="toce" long_name="temperature" unit="degC" grid_ref="Grid_T" enabled="true" />
<field id="toce_K" field_ref="toce" long_name="temperature(K)" unit="degK" />
                                         ↓
<field id="toce_K" long_name="temperature(K)" unit="degK" grid_ref="Grid_T" enabled="true"/>
```

- Warning, reference inheritance is done **AFTER** group inheritance

Disable attribute inheritance by setting its value to “_reset_”

Why Context ?

- Context is similar to "namespace"
- Contexts are isolated from each other, no interference is possible
 - ➡ ids used inside one context can be reused in other context
- For parallelism, each context is associated with its own MPI communicator
 - ➡ No interference between MPI communicators
- Generally a context is associated to one model
 - ➡ Principle of modularity
- A model can declare more than one context

Context element :

- <context>...</context>
- Must be inside of the root XML element
- Must have an id
- Contains calendar and other element definition

```
<context id="nemo" >
  <calendar ... />
  <field_definition> ... </field_definition>
  <file_definition> ... </file_definition>
  <axis_definition> ... </axis_definition>
  <domain_definition> ... </domain_definition>
  <grid_definition> ... </grid_definition>
  <variable_definition> ... </variable_definition>
</context>
```

Each context must define its own calendar

- One calendar by context
- Define a calendar type
 - Date and duration operation are defined with respect to the calendar's type
- Define starting date of the model
- Define time step of the model

Calendar type

- **Gregorian** : standard Gregorian calendar
- **D360** : fixed 360 days calendar
- **NoLeap** : fixed 365 days calendar
- **AllLeap** : fixed 366 days calendar
- **Julian** : Julian calendar (leap every 4 years)
- **user_defined** : months and days can be defined by user (planetology and paleoclimate)

Date and Duration

- A lot of XML attributes are of date or duration type
- Operation between date and duration are strongly dependent of the chosen calendar
 - Ex : date + 1 month = date + 30 day only for month 4,6,9,11

+ Duration units

- Year : **y**
- Month : **mo**
- Day : **d**
- Hour : **h**
- Minute : **mi**
- Second : **s**
- Time step : **ts** (related to time step context definition)

+ Duration format

- Value of unit may be integer or floating (not recommended), mixed unit may be used in a duration definition
 - ➡ Ex. : "**1mo2d1.5h30s**"
 - ➡ Ex. : "**5ts**"

+ Date format

- **year-month-day_hour:minute:second**
 - ➡ Ex. : "**2020-11-04 10:00:00**"
- Partial definition are allowed. Taking into account leftmost part
 - ➡ Ex. "**2020-11**" equivalent to "**2020-11-01 00:00:00**"
 - ➡ Ex. "**2020-11 12**" format error (OK in some case)

Date format

- Date can be also define with a duration offset
 - ➡ Useful for defining a calendar based on standard units (seconds for example)
 - ➡ Ex. : "+3600s"
 - ➡ Or mix : "2012-5 +3600s" equivalent to "2012-5-1 01:00:00"

Attributes for calendar

- **type** : define the calendar type (mandatory)
 - ➡ "Gregorian", "D360", "NoLeap" , "AllLeap", "Julian" or "user_defined"
- **time_origin** : (date) define the simulation starting date ("0000-01-01 00:00:00" by default)
- **start_date** : (date) define the starting date of the run ("0000-01-01 00:00:00" by default)
- **timestep** : (duration) define the time step of the model : mandatory

Setting calendar

- From XML : specific child context element : calendar

```
<context id="nemo" />
  <calendar type="Gregorian" time_origin="2000-01-01" start_date="2020-10" timestep="1h"/>
  ...
</context>
```

Defining an user defined calendar

- Planetology or paleo-climate can not use standard calendar
- Personalised calendar
 - ➡ Defining **day_length** in second (default **86400**)
 - ➡ Defining **month_lengths** : number of days for each month (in an array)

```
<!-- the simplified Darian calendar -->
```

```
<calendar type="user_defined" day_length="88775"  
          month_lengths="(1,24) [28 28 28 28 28 27 28 28 28 28 28 28 27 28 28 28 28 28 28 27 28 28 28 28 28 27]" />
```

➡ Or if you don't want to specify month, you need to define **year_length** in second.

```
<!-- 300 days per year -->
```

```
<calendar type="user_defined" day_length="86400" year_length="25920000"  
          start_date="2020-11 12" />
```

➡ In this way, the format for "date" will no longer contain "month". In Fortran interface, "month"=1

- Possibility to define leap year
 - ➡ Attributes : **leap_year_month**, **leap_year_drift**, **leap_year_drift_offset**
 - ➡ See XIOS user guide



Duration

- Fortran derived type : **TYPE(xios_duration)**
 - (REAL) : **year, month, day, hour, minute, second, timestep**
 - xios_year, xios_month
xios_day, xios_hour
xios_minute
xios_second
xios_timestep

```
TYPE(xios_duration) :: duration
duration%second = 1800
duration = 1800 * xios_second
duration = 0.5 * xios_hour
```

Half an hour

Date

- Fortran derived type : **TYPE(xios_date)**
 - (INTEGER) : **year, month, day, hour, minute, second**

```
TYPE(xios_date) :: date(2014,12,15,10,15,0)
date%year = 2015
```

Date and duration operation

- duration±duration, duration*real, -duration, ==, !=, >, <**
- date-date, ==, !=, >=, >, <=, <**
- date±duration**
- String conversion : **xios_duration_convert_[to/from]_string,**
xios_date_convert_[to/from]_string
- Useful functions : **xios_date_get_second_of_year, xios_date_get_day_of_year,**
xios_date_get_fraction_of_year, xios_date_get_fraction_of_day



Setting calendar from Fortran interface

```
CHARACTER(LEN=*) :: type
TYPE(xios_duration) :: timestep
TYPE(xios_date) :: start_date, time_origin
```

- Within single call

- ➡ **SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin, ...)**
 - ➡ **type** is mandatory.

- Or with individual call

- ➡ **SUBROUTINE xios_set_timestep(timestep)**
 - ➡ **SUBROUTINE xios_set_time_origin(time_origin)**
 - ➡ **SUBROUTINE xios_set_start_date(start_date)**

- calendar **type** must be defined at first.

Hands-on 2-1



 Scalar description : the scalar element <scalar /> Attributes

- (double) **value**
- (string) **name**
- (string) **long_name**
- (string) **scalar_ref**

 More often used in data transformation

- see later

+ Axis description : the axielement <axis />

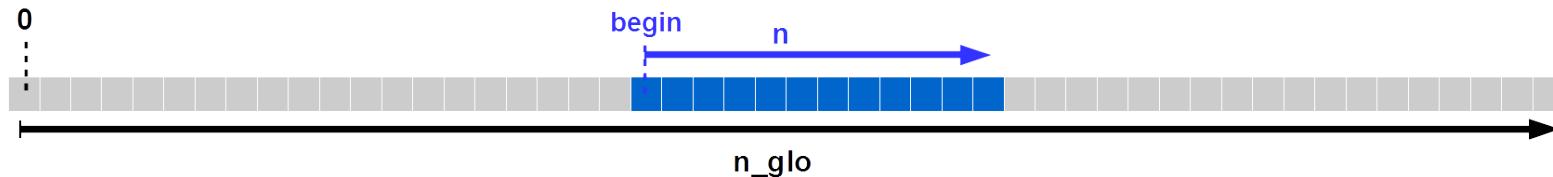
- Describe 1D axis, generally vertical axis
- CALL xios_set_axis_attr("axis_id", ...)

+ Defining the global size of the axis

- (integer) **n_glo** : global size

+ Defining the data parallelism distribution across MPI processes

- (integer) **n** : local axis size distribution
- (integer) **begin** : local axis distribution beginning with respect to the global axis
 - C-convention, starting from 0.
- If nothing specified, the axis is considered as not distributed.
- Data distribution is different for each MPI process, not suitable for XML description
 - Attributes only known at run-time can be passed dynamically using the Fortran interface
 - See section Fortran interface setting attributes



+ Defining axis coordinate values and boundaries

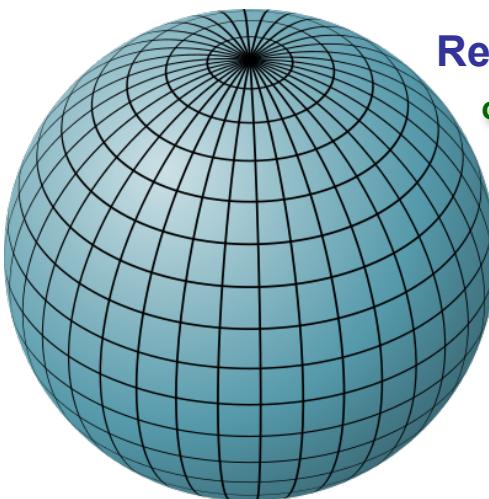
- (real 1D-array) **value[n]**
- (real 2D-array) **bounds[2,n]**

Hands-on 2-2

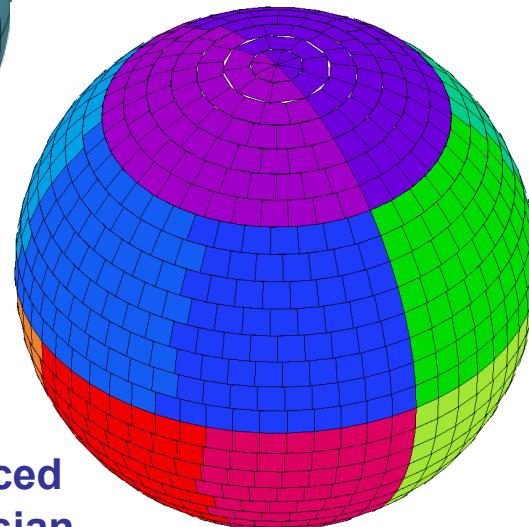


 2D horizontal layer description : the domain element <domain />

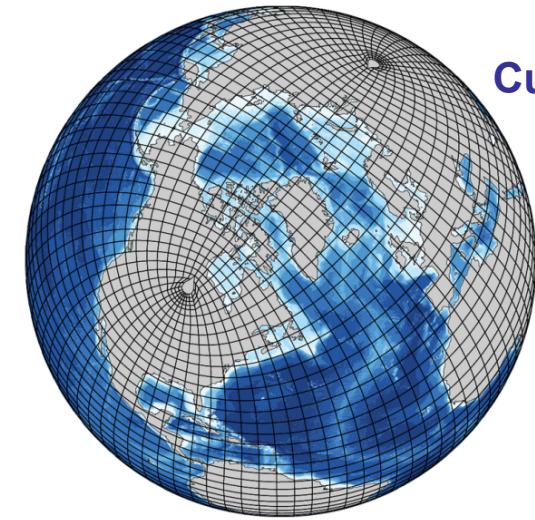
- Describe generally 2D layers mapping the surface of the sphere
- Large variety of 2D domains can be described
- **(string)type :**
 - "rectilinear", "curvilinear", "unstructured", "gaussian"



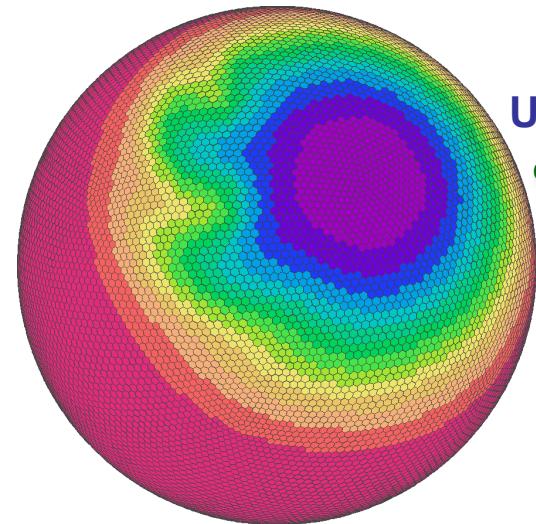
Rectilinear
orchidee & lmdz



Reduced
Gaussian



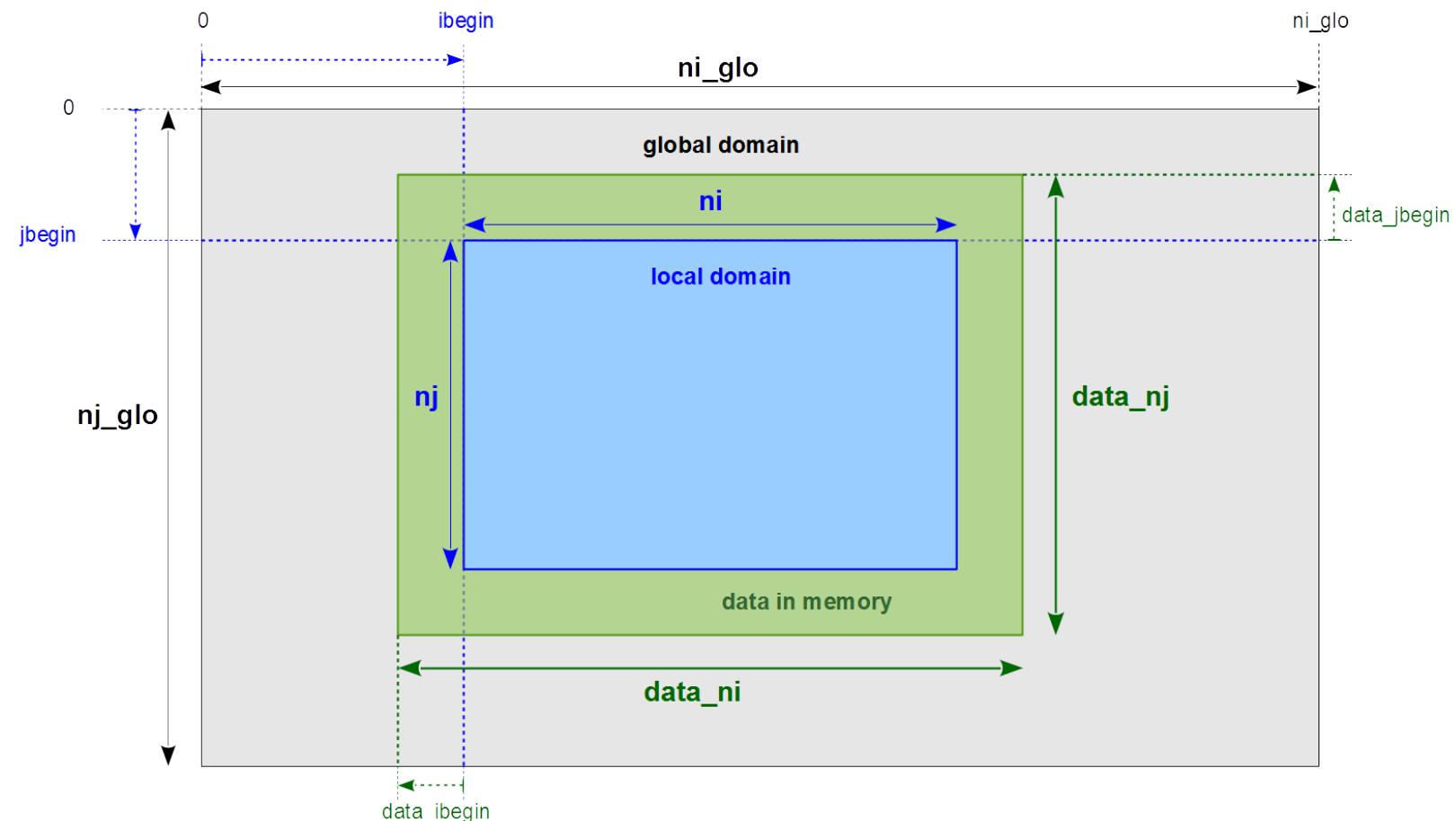
Curvilinear
nemo



Unstructured
dynamico

Rectilinear or curvilinear domains have a 2D description

- (integer) `ni_glo, nj_glo` : global domain size for each direction (longitude and latitude)
- (integer) `ibegin, ni, jbegin, nj` : local domain definition



Defining coordinates

For rectilinear domain

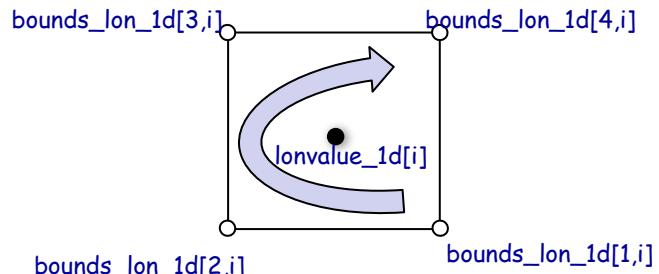
- ▶ **latvalue_1d[nj]** : latitude coordinates of cells
- ▶ **lonvalue_1d[ni]** : longitude coordinates of cells
- ▶ **bounds_lat_1d[4,nj]** : latitudes boundaries of cell corners
- ▶ **bounds_lon_1d[4,ni]** : longitudes boundaries of cell corners

For curvilinear

- ▶ **latvalue_2d[ni,nj]**
- ▶ **lonvalue_2d[ni,nj]**
- ▶ **bounds_lat_2d[4,ni,nj]**
- ▶ **bounds_lon_2d[4,ni,nj]**

For unstructured domain

- ▶ **(integer) nvertex** : max number of corners/edges among cells
- ▶ **(double) latvalue_1d[ni]**
- ▶ **(double) lonvalue_1d[ni]**
- ▶ **(double) bounds_lat_1d[nvertex,ni]**
- ▶ **(double) bounds_lon_1d[nvertex,ni]**



Hands-on 2-3

Describing the mesh : the grid element <grid />

- Can describe element of dimension : 0, 1, ..., 7
- Defined by composition of **scalar**, **axis** and **domain**
- Empty grid is representing a scalar
- 0D : (**scalar**)
- 1D : (**axis**)
- 2D : (**domain**), or (**axis**, **axis**)
- 3D : (**domain**, **axis**), or (**axis**, **axis**, **axis**)
- ...
- recommend using element reference
- can also define element inside

Field geometry is provided by
the underlying mesh description

Can be virtual

```
<grid_definition />

<grid id="grid_3d">
  <domain domain_ref="domain"/>
  <axis axis_ref="axis_Z"/>
</grid >

<grid id="grid_4d">
  <domain id="new_domain" ... />
  <axis id="axis_P" ... />
  <axis id="axis_Q" ... />
</grid >

</ grid_definition />
```

Hands-on 2-4

- + The field element <**field** />
- + Represent incoming or outgoing data flux from models
- + Data can be sent or received at each time step from model through the Fortran interface

- Sending data

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

- Receiving data

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

- + Fields geometry and parallel distribution is hosted by the underlying grid description

- (**string**) **grid_ref** attribute : id of the grid
- For more flexibility fields can refer to a domain
 - ➡ (**string**) **domain_ref** attributes => create a virtual 2D grid composed of the referred domain
 - Or a domain and an axis to create a virtual 3D grid
 - ➡ **domain_ref** and **axis_ref**

```
<grid id="grid_3d">
  <domain id="domain_2d"/>
  <axis id="axis_1d" />
</grid>
...
<field id="temp" grid_ref="grid_3d"/>
```

~

```
<axis id="axis_1d" />
<domain id="domain_2d"/>
...
<field id="temp" domain_ref="domain_2d"
      axis_ref="axis_1d"/>
```

Field data from models must be conform to the grid description

- Fields can be declared of any dimensions in single or double precision
- But total **size** and **data order** must be the same as declared in the grid

→ Example :

```
<grid id="grid_3d">
  <domain id="domain_2d" type="rectilinear" ni_glo="100" ni="10" data_ni="12"
        nj_glo="50" nj="5" data_nj="7"/>
  <axis id="axis_1d" n_glo="20"/>
</grid>
...
<field id="temp" grid_ref="grid_3d"/>
```

- Global grid : 100x50x20
- Local grid : 10x5x20
- Data in model memory : **data_ni × data_nj × n_glo = 12×7×20 = 1680**
- Can be declared as :
 - **REAL(kind=4) :: temp(12,7,20)**
 - **REAL(kind=4) :: temp(1680)**
 - **REAL(kind=8) :: temp(1680)**
- but data order follows the column major order Fortran convention

Field can be output to files

- Will appear as a child element of **file** element
- A field can appear, in multiple files
 - using the reference attribute : **field_ref**

```
<field_definition>
    <field id="temp"  grid_ref="grid_3d"/>
    <field id="precip"  grid_ref="grid_3d"/>
    <field id="pressure" domain_ref="domain_2d"/>
</field_definition>

<file_definition>
    <file name="daily_output" freq_output="1d">
        <field field_ref="temp" />
        <field field_ref="pressure" />
    </file>

    <file name="monthly_output" freq_output="1mo">
        <field field_ref="temp" />
        <field field_ref="precip" />
    </file>
</file_definition>
```

Field attributes

● Field description :

- ➡ (string) **name** : name of the field in the file. If not specified, "**id**" will be used in place
- ➡ (string) **long_name** : set "long_name" netcdf attribute conforming to CF compliance
- ➡ (string) **standard_name** : set "standard_name" netcdf attribute
- ➡ (string) **unit** : set "unit" netcdf attribute
- ➡ (double) **valid_min/valid_max** : set **valid_min** & **valid_max** netcdf attribute

● Enable/disable field output :

- ➡ (boolean) **enabled** : if **false**, field will not be output (**default=true**)
- ➡ (integer) **level** : set the output level of the field (**default=0**) with respect to the file attribute "**level_output**". If (**level>level_output**) the field will not be output.

● Precision and compression :

- ➡ (integer) **prec** : define the output precision of the field : **8->double, 4->single, 2->2-byte integer**
- ➡ (double) **add_offset, scale_factor** : output will be **(field+add_offset)/scale_factor**
- ➡ (integer) **compression_level (0-9)** : set the gzip compression level provided by netcdf4/hdf5: due to HDF5 limitation, doesn't work for parallel writing. If not set data is not compressed.
- ➡ (boolean) **indexed_output** : if set to **true**, only non masked value are output.

Field time integration

- At each time step , data field are exposed from model (**xios_send_field**)
- Data are extracted according to the grid definition
- Time integration can be performed on incoming flux
- The time integration period is fixed by file output frequency (**output_freq** attribute)
- **(string) operation** attribute : time operation applied on incoming flux
 - ➡ **once** : data are used one time (first time)
 - ➡ **instant** : instant data values will be used
 - ➡ **maximum** : retains maximum data values over the integration period
 - ➡ **minimum** : retains minimum data values over the integration period
 - ➡ **average** : make a time average over the period
 - ➡ **cumulate** : cumulate date over the period
- Example : each day, output the time average and instant values of "temp" field

```
<file name="output" output_freq="1d">
  <field field_ref="temp" name="temp_average" operation="average"/>
  <field field_ref="temp" name="temp_instant" operation="instant"/>
</file>
```



Time sampling management

- Some field are not computed every time step
- (duration) **freq_op** attribute: field will be extract from model at "**freq_op**" frequency
- (duration) **freq_offset** attribute: time offset before extracting the field at "**freq_op**" frequency
- Strongly advised to set **freq_op** and **freq_offset** as a multiple of time step
- Example : for making a daily averaging, get "**temp**" value every 10 time step. The first value extracted will be at 2nd time step.

```
<file name="output" output_freq="1d">
  <field field_ref="temp" operation="average" freq_op="10ts" freq_offset="1ts"/>
</file>
```

Undefined values and time operation

- Undefined values must not participate to time integration operation
 - ➡ Set **default_value** attribute as the undefined value (missing value). If not defined, missing value will be 0.
 - ➡ (boolean) **detect_missing_value** : for the current time step, all field value equal to **default_value** (undefined value) will not be taking into account to perform the time integration (**average**, **minimum**, **maximum**, **cumulate**)
- Very expensive since each value of the mesh must be tested

Hands-on 2-5

+ Output file : the file element <file />

+ Defining fields to be written

- File elements can contains **field** elements or **field_group** elements
- All listed field elements are candidates for output
- (string) **field_group_ref** attribute: fields included in the referred field group will be included in file

```
<field_definition>
  <field_group id="fields_3d" grid_ref="grid_3d">
    <field id="temp"  >
    <field id="precip" >
  </field_group>
  <field id="pressure" domain_ref="domain_2d"/>
</field_definition>

<file_definition>
  <file name="daily_output" freq_output="1d">
    <field_group group_ref="fields_3d" operation="average"/>
    <field_group operation="instant"/>
      <field field_ref="temp"   name="temp_inst" />
      <field field_ref="pressure" name="pressure_inst" />
    </field_group>
    <field field_ref="pressure" operation="average" />
  </file>
</file_definition>
```

- Variables output as average :
 - temp
 - precip
 - pressure

- Variables output as instant
 - temp_inst
 - pressure_inst

- Variables output as instant
 - temp_inst
 - pressure_inst

Enabling /disabling output

- Field can be enabled/disabled individually
 - ➡ (bool) **enabled** field attribute
- Enable/disable with **level** output
 - ➡ (integer) **output_level** file attribute : set level of output
 - ➡ (integer) **level** field attribute : if **level** > **output_level**, field is disabled
- Enable/disable all fields
 - ➡ (bool) **enabled** file attribute : if set to **false**, all fields are disabled
- Files with all fields disabled will not be output

Hands-on 2-6

File format

- For now file output format is only **NETCDF**
 - ➡ **Grib2** and **HDF5** output format will be considered in future
- Can choose between parallel write into a single file or multiple file (1 file by xios server)
 - ➡ (string) **type** attribute : select output mode "**one_file**" / "**multiple_file**"
 - ➡ For "**multiple_file**" mode, files are suffixed with xios servers ranks
- Can choose between **netcdf4** et **netcdf4 classical** format
 - ➡ (string) **format** attribute : "**netcdf4**" for **netcdf4/hdf5** or "**netcdf4_classical**" for historical **netcdf3** format
 - ➡ In "**one_file**" mode, use **hdf5** parallel for **netcdf4** format and **pnetcdf** for classical format.
 - ➡ Sequential **netcdf** library can be used in **multiple_file** mode
- Data can be compressed : only available with **netcdf4** format (**hdf5**) in sequential write (**multiple_file**)
 - ➡ (integer) **compression_level** attribute : compression level (**0-9**), can be fixed individually with field attribute

Setting parameters : the variable element <variable/>

- Variable are used to define parameters
- Variable can be set or queried from model
 - Could replace Fortran namelist or IPSL run.def files
- Used internally by XIOS to define its own parameters

Attributes

- **(string) name** : name of the attribute (optional)
- **(string) type** : type of the variable (optional)
 - "bool", "int16", "int", "int32", "int64", "float", "double", "string"

Setting variable values from XML

- Values are defined in the content section

```
<file>
  <variable id="int_var" type="int"> 10 </variable>
  <variable id="string_var" type="string">a string variable</variable>
</file>
```

variable_definition and variable_group

Set or query value from model

- Set variable : **ierr = xios_setvar('var_id',variable)**
- Get variable : **ierr = xios_getvar('var_id',variable)**
 - Return **true** if 'var_id' is defined and second argument contains the read value
 - return **false** if 'var_id' is not defined and second argument value is unchanged

```
<variable_definition>
    <variable id="int_var" type="int"/> 10 </var>
    <variable id="string_var" type="string">a string variable</variable>
</variable_definition>
```

USE xios

...

INTEGER :: int_var

CHARACTER(LEN=256) :: string_var

LOGICAL :: ierr

```
ierr=xios_getvar('int_var',intvar)
ierr=xios_setvar('int_var',intvar+2)
ierr=xios_getvar('int_var',intvar)      ! -> int_var=12
ierr=xios_getvar('string_var',string_var) ! -> string_var="a string variable"
```

Defining how data are stored in memory

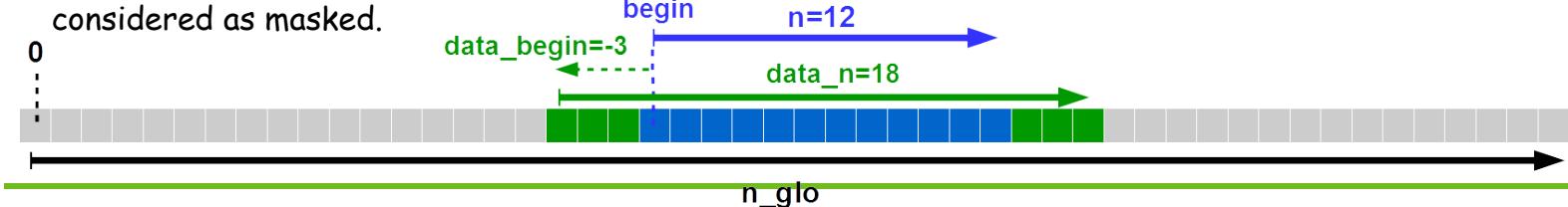
- Data are stored in memory as Fortran array
- But data can be masked, or ghost cells are not valid data, or axis value can be compressed
- XIOS will extract only required value from memory
- Must describe valid data with attributes
- Whole data are valid by default

Masking Data (optional)

- (boolean 1D-array) **mask[n]** (false/zero : data masked)
- Masked data will not be extracted from memory and will appear as missing values in output files

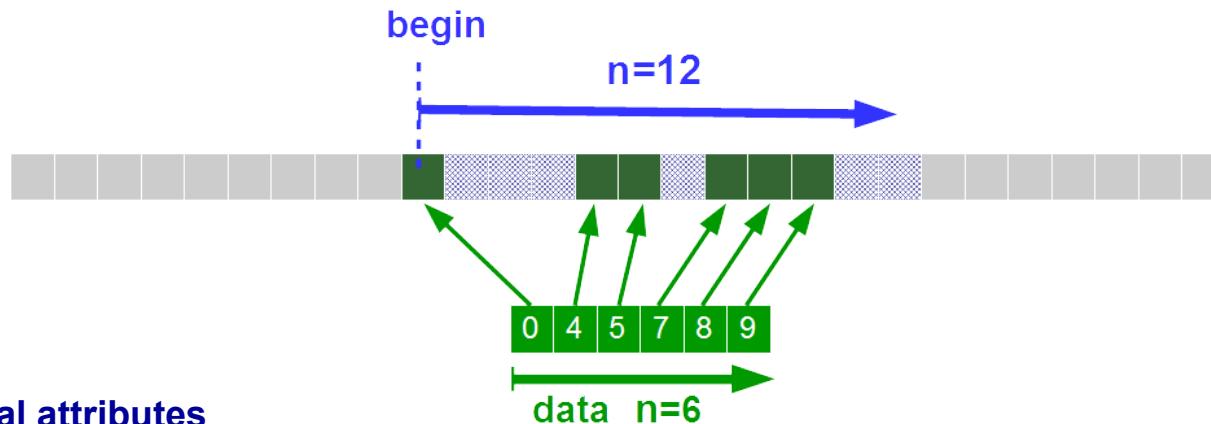
Defining ghost cells (optional)

- (integer) **data_n** : size of the data in memory (default : **data_n=n**)
- (integer) **data_begin** : offset with respect to local axis distribution beginning
 - ➡ default : **data_begin=0**
 - ➡ Negative offset : data outside of the local distribution will not be extracted (ghost cell)
 - ➡ Positive offset : data in interval [**begin**, **data_begin**] and/or [**data_begin+data_n-1**, **begin+n-1**] are considered as masked.



Defining compressed data (optional)

- Data can be compressed in memory (ex : land point), and can be decompressed for output
- Undefined data are considered as masked and will be output as missing value
- (integer 1D-array) **data_index**
 - Define the mapping between data in memory and the corresponding index into the local axis distribution
 - **data_index[i]=0** map the beginning of the local distribution
 - Negative index or greater than **n-1** will be outside of the distribution and will not be extracted



Other optional attributes

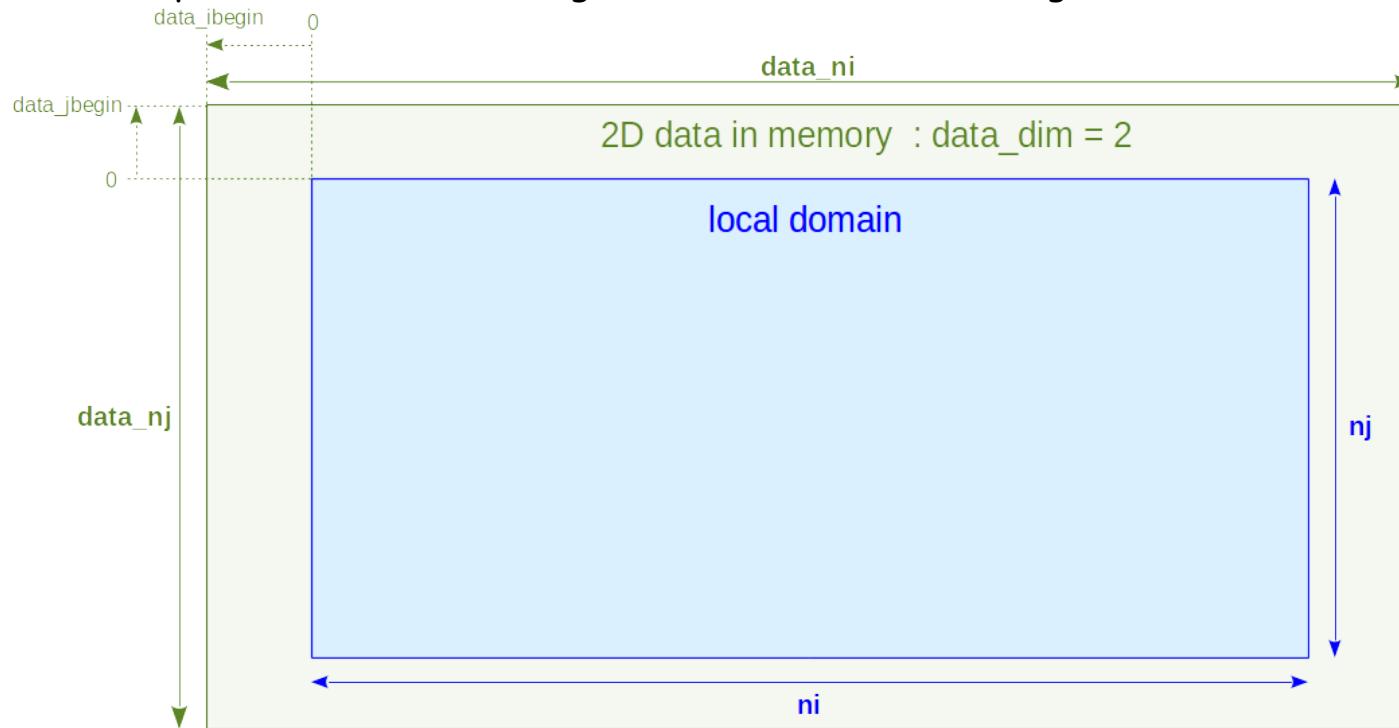
- (string) **name**
- (string) **long_name**
- (string) **unit**
- (bool) **positive** : set "positive" CF attribute in Netcdf output

Masking grid point individually

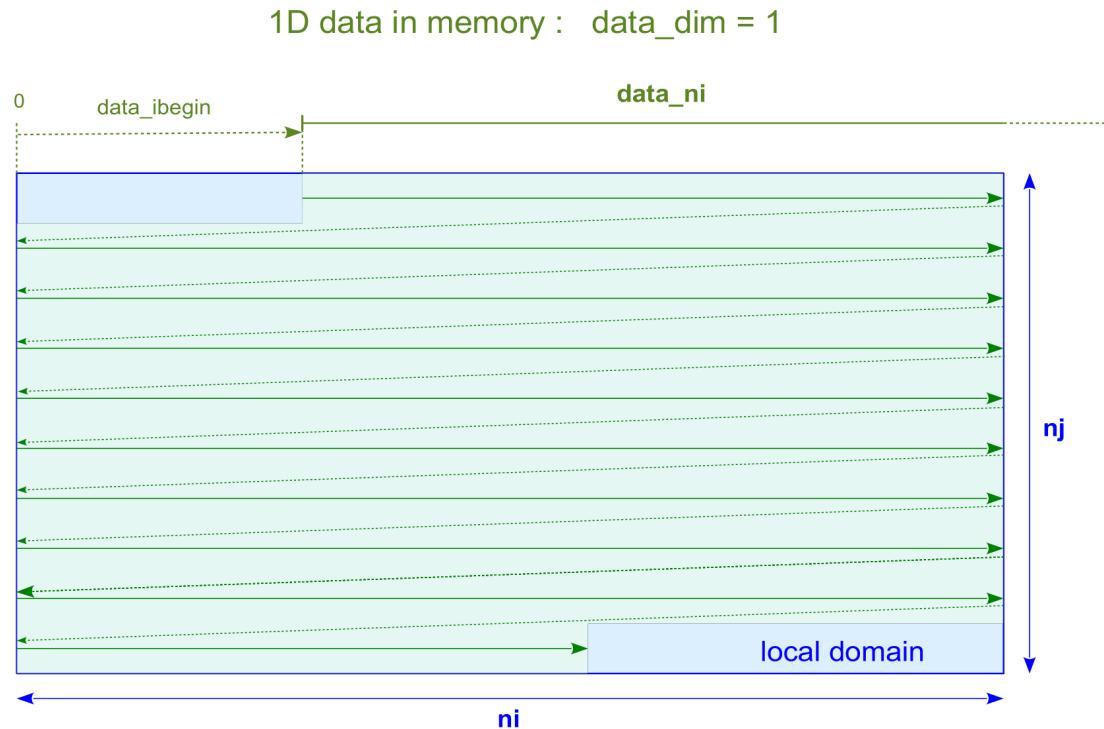
- Masking one point in the 3rd axis means masking a full 2D layer in the 3d grid
- Grid point can be masked using the mask attribute
- Regarding of the dimensionality of mask arrays, version mask_1d to mask_7d are allowed
 - ➡ Total mask size must be equal to the local domain size
 - ➡ Ex : <grid id="grid_3d" mask_3d="(0,9)x(0,4)x(0,19)[0 1 1 0 ... 0 1]">
 - ➡ or <grid id="grid_3d" mask_1d="(0,9990)[0 1 1 0 ... 0 1]">
 - ➡ Not practical with xml. Better set mask via Fortran API.

Data representation in memory : similar to 1D-axis but for 2 dimensions

- Can be 1D-array (horizontal layer as a vector) or 2D-array
 - ➡ (integer) **data_dim** attribute : 1(default) or 2
- (integer) **data_ni** : size of the first array dimension
- (integer) **data_ibegin** attribute : Offset for the first dimension with respect to local domain distribution beginning : may be negative or positive (default : **data_ibegin=0**)
- [if **data_dim=2**] **data_nj**, **data_jbegin** : 2nd dimension (default: **data_nj=nj**, **data_jbegin=0**)
- Example for **data_dim=2**, negative offsets to eliminate ghost cells



- Example for **data_dim=1** : horizontal layer seen as a vector
 - Positive offsets, local domain from different processes can overlap



Unstructured domain has a 1D description

- Vector of cells
 - ➡ `ni_glo`, `ni` and `ibegin` can be specified
 - ➡ `nj_glo`, `nj` and `jbegin` are meaningless
- Data in memory is always a vector
 - ➡ `data_dim=1`

Compressed data (on “data”)

- For `data_dim=1` (decompressed data is a 1D-array)
 - ➡ `data_i_index[data_ni]` : index for decompressed local domain represented by vector (exclusive with `data_ibegin`)
- For `data_dim=2` (decompressed data is a 2D-array)
 - ➡ `data_nj` must be equal to `data_ni`
 - ➡ `data_i_index[data_ni]`, `data_j_index[data_ni]` : indexes for decompressed local domain represented as a 2D-array (exclusive with `data_ibegin`, `data_jbegin`)

Masking data (on “grid”)

- (boolean 1D-array) `mask_1d` attribute : 1d array version
 - ➡ `mask_1d[ni*nj]` for rectilinear and curvilinear domain
 - ➡ `mask_1d[ni]` for unstructured
- (boolean 2D-array) `mask_2d` attribute : 2d array version
 - ➡ `mask_2d[ni,nj]` for rectilinear and curvilinear domain only

Hands-on 3

File structure

- XIOS respects CF convention as much as possible
- One time record (unlimited dimension) by file
 - ➡ (duration) **output_freq** attribute : define the output frequency and the time axis
 - ➡ **time_counter** dimension and axis are written conforming to CF convention
- Can mix instant and average time operation
 - ➡ Axis **time_instant** or **time_centred** may be written with the associated bounds
- Fields of different grids can be in same file
 - ➡ Longitude, latitude and verticals axis are automatically written with the associate metadata following CF convention
 - ➡ Axis boundaries will be also written if available
- Some fields attributes (**standard_name**, **long_name**, **unit**,...) will be output as field metadata

● Example of netcdf file output with XIOS

```

netcdf output_atmosphere_2D_HR {
dimensions:
    axis_nbounds = 2 ;
    lon = 200 ;
    lat = 200 ;
    time_counter = UNLIMITED ; // (30 currently)
variables:
    float lat(lat) ;
        lat:axis = "Y" ;
        lat:standard_name = "latitude" ;
        lat:long_name = "Latitude" ;
        lat:units = "degrees_north" ;
        lat:nav_model = "domain_atm_HR" ;
    float lon(lon) ;
        lon:axis = "X" ;
        lon:standard_name = "longitude" ;
        lon:long_name = "Longitude" ;
        lon:units = "degrees_east" ;
        lon:nav_model = "domain_atm_HR" ;
    float tsol(time_counter, lat, lon) ;
        tsol:long_name = "Surface Temperature" ;
        tsol:online_operation = "average" ;
        tsol:interval_operation = "3600 s" ;
        tsol:interval_write = "1 d" ;
        tsol:cell_methods = "time: mean (interval: 3600 s)" ;
        tsol:coordinates = "time_centered" ;
    double time_centered(time_counter) ;
        time_centered:standard_name = "time" ;
        time_centered:long_name = "Time axis" ;
        time_centered:calendar = "gregorian" ;
        time_centered:units = "seconds since 1999-01-01 15:00:00" ;
        time_centered:time_origin = "1999-01-01 15:00:00" ;
        time_centered:bounds = "time_centered_bounds" ;
    double time_centered_bounds(time_counter, axis_nbounds) ;
    double time_counter(time_counter) ;
        time_counter:axis = "T" ;
        time_counter:standard_name = "time" ;
        time_counter:long_name = "Time axis" ;
        time_counter:calendar = "gregorian" ;
        time_counter:units = "seconds since 1999-01-01 15:00:00" ;
        time_counter:time_origin = "1999-01-01 15:00:00" ;
        time_counter:bounds = "time_counter_bounds" ;
    double time_counter_bounds(time_counter, axis_nbounds) ;

// global attributes:
    :name = "output_atmosphere_2D_HR" ;
    :description = "Created by xiōs" ;
    :title = "Created by xiōs" ;
    :Conventions = "CF-1.5" ;
    :production = "An IPSL model" ;
    :timeStamp = "2015-Dec-14 15:20:26 CET" ;

```



+ Adding specific metadata

- Using variable element <**variable**>
- Variable as file child will be output as a global **netcdf** file attribute
- Variable as field child will be output as a **netcdf** variable attribute
- Example :

```
<file name="daily_output" freq_output="1d">
  <field field_ref="pressure" operation="average" >
    <variable name="int_attr" type="int"> 10 </variable>
    <variable name="double_attr" type="double"> 3.141592654 </variable>
  </field>
  <variable name="global_attribute" type="string"> A global file attribute </variable>
</file>
```

+ Flushing files

- File can be flushed periodically in order to force data in cache to be written
- (**duration**) **sync_freq** file attribute : flush file at **sync_freq** period

Appending data to an existing file

- When restart models, field data can be appended to a previous XIOS output file
- (bool) **append** attribute : if set to **true** and if file is present, data will be appended
 - Otherwise a new file will be created
 - Default is creating a new file (**append=false**)

Splitting files

Hands-on 4

- In order to avoid big file, file can be split periodically
- File suffixed with start date and end date period
- (duration) **split_freq** : split file **at split_freq** period

Generating time series (CMIP requirement)

- Fields included into a single file may be automatically spread into individual files
- One field by file, file name based on field name
 - (string) **ts_prefix** file attribute : prefix for time series files
 - (bool) **ts_enabled** field attribute : is set to true, field is candidate to be output as time series
 - (duration) **ts_split_freq** field attribute: individual field split frequency (default is file splitting frequency)
- (string) **timeseries** file attribute (**none / only / both / exclusive**) : activate time series output
 - none** : standard output, no time series
 - only** : only field **with ts_enabled="true"** will be output as time series and no other output
 - both** : timeseries + full file
 - exclusive** : field **with ts_enabled="true"** will be output as time series, the other field in a single file

Reading data from file

- (string) **mode** attribute ("read" / "write") : if set to read, file will be an input
- Each time record will be read at every **freq_output** frequency (a little ambiguous but ...)
- Value can be get from models at the corresponding time step using :
`CALL xios_recv_field("field_id", field)`
- First time record will sent to model at time step 0 (before time loop).
- Except using **freq_offset** field attribute
 - Exemple : `freq_offset="1ts"` : first record will be read at first time step and not 0

--- xml ---

```
<file name="daily_output" freq_output="1ts" mode="read" >
  <field id="temp" operation="instant" freq_offset="1ts" grid_ref="grid_3d"/>
</file>
```

--- model ---

```
DO ts=1,n
  CALL xios_update_timestep(ts)
  CALL xios_recv_field("temp",temp)
ENDDO
```

Hands-on 5

- Field with no time record will be read only once



Why Workflow ?

- Field are exposed from model at each time step
 - ➡ internally representing data flux assigned to a timestamp
- Each data flux can be connected to one or more filters
- Filters are connected to one or more input flux and generate a new flux on output
- All filters can be chained together to achieve complex operations
- All filters are parallel
- XML file describe a full graph of parallel tasks

+ Workflow entry point

- Input flux can be a field sent from model (`xios_send_field`)
- Input flux can be a field read from an input file (`mode="read"`)

+ Workflow end point

- Output flux can be sent to servers and written to file (`mode="write"`)
- Output flux can be read from model (`xios_recv_field`)
 - ➡ `(bool) read_access` field attribute : field read from models must set `read_access="true"`
 - ➡ Field read from file have automatically `read_access="true"`



```
--- xml ---
<field id="precip" grid_ref="grid_3d"/>
<field id="pressure" field_ref="p" read_access="true" unit="Pa" / >
<field id="precip_read" field_ref ="precip" read_access="true" />

<file name="daily_output" freq_output="1ts">
  <field id="temp" operation="instant" grid_ref="grid_3d"/>
  <field id="p"  operation="instant" domain_ref="domain_2d"/>
</file>

--- model ---
DO ts=1,n
  CALL xios_update_timestep(ts)
  CALL xios_send_field("precip",precip)
  CALL xios_send_field("temp",temp)
  CALL xios_recv_field("pressure",pressure)
  CALL xios_recv_field("precip_read",precip_read) ! Now precip_read==precip
ENDDO
```

field_ref attribute : duplicate flux from the referenced field

- For each reference to field, a new flux is created by duplicating source flux

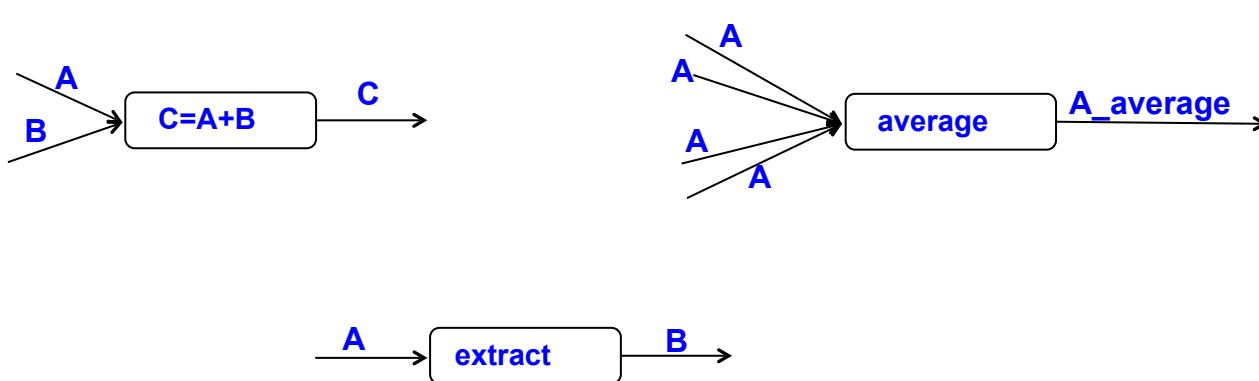


- Also, make XML inheritance

Defining filters and transformations

Actually 3 kinds of filters

- Arithmetic filters : combine flux together
- Temporal filters : integrate flux over a period of time
- Spatial filters : transform the geometry of the incoming flux



Arithmetic filters

- Arithmetic filter can combine different flux of same timestamp with arithmetic operator or function
- All incoming flux must be on the same grid
 - Perform same operations for each grid point
- Arithmetic filter are defined in the content section of a field element
- Computed flux value will replace actual flux, even if coming from reference

```
<field id="temp" unit="°C" grid_ref="grid_3d"/>
<field id="temp_K" unit="°K" field_ref="temp"> temp+273.15 </field>
```

- Specific "this" (auto-reference) keyword representing the incoming flux of the current field

```
<field id="temp" unit="°K" grid_ref="grid_3d"> this+273.15 </field>
```

- Arithmetic filters can be easily chained,

→ Computed flux can be reused

$$C = \frac{A + B}{A * B}$$

$$D = \frac{e^{-C*D}}{3}$$

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

Time integration filters

Time filters of are specified with the "operation" field attribute

- Possible value : "once", "instant", "maximum", "minimum", "average", "accumulate"
- A new flux is generated at the end of the time integration period

Time filter is enabled only if :

- Field is included into a file
 - ➡ **output_freq** define the period over which integration is done
 - ➡ Generated flux is sent to server to be recorded
- Flux can be reused by an other field after time integration
 - ➡ The @ operator : means that time integration is performed over the flux
 - ➡ The time integration period is given by value of **freq_op** attribute of new flux

```
<field id="temp" operation="average" />
<field id="temp_ave" freq_op="1d"/> @temp </field>
```

- ➡ New flux "temp_ave" is created every day (**freq_op="1day"**) by time averaging of "temp" flux

⊕ Chaining time filters

- Using the @ operator
- Example : compute and output the monthly average of the daily maximum and minimum of temperature and the monthly maximum and minimum of the daily temperature average

--- xml ---

```
<field id="temp"          operation="average"/>
<field id="temp_min" field_ref="temp" operation="minimum" />
<field id="temp_max" field_ref="temp" operation="maximum" />

<file name="monthly_output" output_freq="1mo" />
  <field name="ave_daily_min" operation="average" freq_op="1d"> @temp_min </field>
  <field name="ave_daily_max" operation="average" freq_op="1d"> @temp_max </field>
  <field name="min_daily_ave" operation="minimum" freq_op="1d"> @temp </field>
  <field name="max_daily_ave" operation="maximum" freq_op="1d"> @temp </field>
</file>
```

2nd

1st

--- model ---

```
CALL xios_send_field("temp", temp)
```

⊕ Chaining and combine time filters and arithmetic's filters

- Compute the time variance of a temperature field $\sigma \approx \sqrt{\langle T^2 \rangle - \langle T \rangle^2}$

--- xml ---

```
<field id="temp" operation="average"/>
<field id="temp2" field_ref="temp" /> temp*temp </field>

<file name="monthly_output" output_freq="1mo" />
  <field name="sigma_T" operation="instant" freq_op="1mo"> sqrt(@temp2-pow(@temp,2)) </field>
</file>
```

--- model ---

```
CALL xios_send_field("temp",temp)
```

Hands-on 6

Spatial filters

- Spatial filters may change the geometry, dimensionality and the parallelism data distribution of a flux
- Algorithms must be parallel and scalable in order to perform the flux transformation on whole allocated parallel resources of a simulation
- More filters under development

Using spatial filter

- Spatial filters are enabled when the grid of a referenced field is different of the current grid field

```
<field id="temp" grid_ref="grid_regular"/>
<field id="new_temp" field_ref="temp" grid_ref="grid_regular" />
```

→ No spatial filter enabled
(same grid ref)

```
<field id="temp" grid_ref="grid_regular"/>
<field id="new_temp" field_ref="temp" grid_ref="grid_unstruct" />
```

→ Trigger spatial filter
(different grid ref)

- If grid are not matching exactly, try to find a way to transform source grid into target grid
 - If not possible an error is generated
 - Otherwise filter will be used

- To find which filter to activate, a matching is done between domain and axis composing the grid.
 - An exact matching between element do not activate filter
 - If not matching, see if it is possible to transform the source element domain or axis into target element with a transformation.
 - Otherwise an error is generated

```
<axis id="vert_axis" n_glo="100" />
<domain id="regular" ni_glo="360" nj_glo="180" type="rectilinear" />
<domain id="unstruct" ni_glo="10000" type="unstructured" />

<grid id="grid_regular">
  <domain domain_ref="regular">
    <axis axis_ref="vert_axis" >
  </grid>

<grid id="grid_unstruct">
  <domain domain_ref="unstructured">
    <interpolate_domain/>
  <domain/>
    <axis axis_ref="vert_axis" >
  </grid>

<field id="temp" grid_ref="grid_regular"/>
<field id="new_temp" field_ref="temp" grid_ref="grid_unstruct" />
```

- More than one filter can be implemented in same transformation

```
<axis id="vert_src" n_glo="100" />
<axis id="vert_dst" n_glo="50" />

<domain id="regular" ni_glo="360" nj_glo="180" type="rectilinear" />
<domain id="unstruct" ni_glo="10000" type="unstructured" />

<grid id="grid_regular">
  <domain domain_ref="regular"/>
  <axis axis_ref="vert_src" />
</grid>

<grid id="grid_unstructured">
  <domain domain_ref="unstructured">
    <interpolate_domain/>
  </domain>
  <axis axis_ref="vert_dst">
    <interpolate_axis/>
  </axis>
</grid>
```

- Domain interpolation will be perform first "regular" -> "unstructured"
- Axis interpolation will be perform in 2nd time "vert_src" -> "vert_dst"

Available spatial filters :

Extract

- Extract sub-part of data : `extract_axis`, `extract_domain`
- Extract axis to scalar
 - ➡ **(integer) position** : position of the element to be extract from axis.
- Extract axis to axis
 - ➡ **(integer) begin** : begin position of the element to be extract from axis.
 - ➡ **(integer) n** : number of elements to be extract from axis.
 - ➡ **(1D-array) index** : array including all indexes of elements to be extract from axis.
- Extract domain to axis
 - ➡ **(string) direction** : "iDir" or "jDir"
 - ➡ **(integer) position** : position of the slice to be extract from domain.
- Extract domain to domain
 - ➡ **(integer) ni** : number of elements to be extract from domain along the i-direction.
 - ➡ **(integer) nj** : number of elements to be extract from domain along the j-direction.
 - ➡ **(integer) ibegin** : i-position of starting element to be extract from domain.
 - ➡ **(integer) jbegin** : j-position of starting element to be extract from domain.

```
<domain id="regular" ni_glo="360" nj_glo="180" type="rectilinear" />
<axis id="axis" n_glo="100" />

<grid id="grid_src">
  <domain domain_ref="regular"/>
  <axis axis_ref="axis"/>
</grid>

<grid id="grid_extract">
  <domain domain_ref="regular">
    <extract_domain ibegin="20" ni="50" jbegin="100" nj="60" />
  </domain/>
  <axis axis_ref="axis">
    <extract_axis begin="30" n="10"/>
  </axis>
</grid>

<field id="field" grid_ref="grid_src"/>
<field id="field_extracted" field_ref="field" grid_ref="grid_extract" />
```

- ➡ Extract data of size **(50,60,10)** starting at index **(20,100,30)**
- ➡ Only the extracted part will be output to files

Hands-on 7-1

Hands-on 7-2



Available spatial filters :

Reduce

- Reduce data : **reduceac_scalar**, **reduce_axis**, **reduce_domain**

Hands-on 7-3

- Reduce scalar to scalar

 ➡ **(string) operation** : sum, average, max, min. Perform a MPI-Reduce

Hands-on 7-4

- Reduce axis to scalar

 ➡ **(string) operation** : sum, average, max, min.

- Reduce axis to axis

 ➡ **(string) operation** : sum, average, max, min. Perform a MPI-Reduce

- Reduce domain to scalar

 ➡ **(string) operation** : sum, average, max, min.

 ➡ **(bool) local** : whether the reduction should be performed locally on data owned by each process or on the global domain (default "false")

- Reduce domain to axis

 ➡ **(string) operation** : sum, average, max, min.

 ➡ **(string) direction** : "iDir" or "jDir"

 ➡ **(bool) local** : whether the reduction should be performed locally on data owned by each process or on the global domain (default "false")

 Inverse

- **inverse_axis**

 Duplicate

- **duplicate_scalar** : duplicate scalar to axis

 Reorder

- **reorder_domain** : duplicate scalar to axis

→ (bool) **invert_lat** : define whether the latitude should be inverted. (default "false")

→ (double) **shift_lon_fraction** : longitude offset. Represents a fraction of **ni_glo**. (default "0")

→ (double) **max_lon** : optional.

→ (double) **min_lon** : optional.

→ If both **min_lon** and **max_lon** are defined, domain will be reordered with latitude values ranging from **min_lon** to **max_lon** .

Hands-on 7-5

Generate domain

- **Generate_rectilinear_domain**

- (double) **lon_start, lon_end, lat_start, lat_end**
- (double) **bounds_lon_start, bounds_lon_end, bounds_lat_start, bounds_lat_end**
- Range in [0°, 360°] for longitude, [-90°, 90°] for latitude
- Useful to perform automatic interpolation on regular grid
- Generate automatically parallel distribution, longitude and latitude values
- **ni_glo** and **nj_glo** must be defined in the domain element

Interpolate (only polynomial)

● **interpolate_domain**

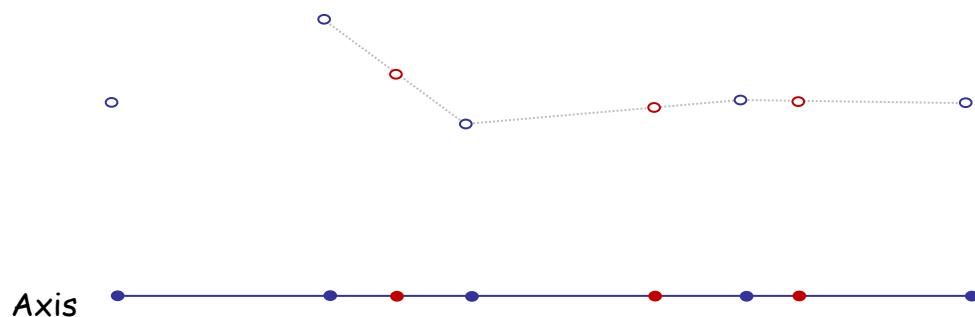
- ➡ Perform interpolation between any kind of domain
 - ➡ Compute weight on the fly and in parallel at XIOS closing definition step
 - ➡ Interpolation is done on parallel on the incoming distributed flux
 - ➡ Current algorithm is only conservative remapping of 1st or 2nd order
-
- ➡ **(integer) order** : set the order (1 or 2) of the conservative interpolation (default "2")
 - ➡ **(bool) renormalize** : used in case where targeted cells intersect masked source cells. If set to "true", flux is renormalized prorate of the non masked intersected area. (default "false")
 - ➡ **(bool) quantity** : set to "true" to preserve extensive property of the field (default "false")
 - ➡ **(bool) detect_missing_value** : if set to "true" , input data of the field to be interpolated are analyzed to detect missing values. (default "false")
 - ➡ **(bool) use_aera** : if set to "true" , area for source and target domain (if any) will be used to renormalize compute weight by the ratio given area / computed area. Default value is false. Used with domain **radius** attribute
 - ➡ **(string) mode** : "read", "compute", "read_or_compute". This attribute determines the way to obtain interpolation weight information. Default "compute"
 - ➡ **(bool) write_weight** : set to "true" to write the computed weight to file.
 - ➡ **(string) weight_filename** : define the file name where the weights will be written or read. If not specified, when trying to read or write, a name will be automatically generated (contextid_srcdomain_destdomain).
 - ➡ **(string) read_write_convention** : index will begin from 0 if set to "c", from 1 if set to "fortran"

Hands-on 7-6

Interpolate (only polynomial)

● interpolate_axis

- ➡ (integer) **order** : optional. set the order of the polynomial interpolation (default “1”)
- ➡ (string) **type** : “polynomial” only. Optional
- ➡ (string) **coordinate** : defines the coordinate (**value**) associated with an axis on which interpolation will be performed
- ➡ Apply only on 3D field



Hands-on 7-7

Chaining spatial transformation

- Chaining can be easily achieved by referencing intermediate field

→ Ex : interpolate unstructured grid to regular and then make a zoom

```
<field id="temp_unstr"          grid_ref="grid_unstruct"/>
<field id="temp_reg"      field_ref="temp_unstr" grid_ref="grid_regular"/>
<field id="temp_reg_extract" field_ref="temp_reg"  grid_ref="grid_regular_extract"/>
```

- To avoid intermediate field definition, use **grid_path** attribute

→ (string) **grid_path** attribute : define the list of intermediate grid (**grid_path="grid1,grid2"**)

```
<field id="temp_unstr"      grid_ref="grid_unstruct"/>

<field id="temp_reg_extract" field_ref="temp_unstr" grid_path="grid_regular"
       grid_ref="grid_regular_extract"/>
```

- Other possibilities is to chain transformation in domain or axis definition

```
<field id="temp_unstr"          domain_ref="unstructured" />
<field id="temp_reg_extract" field_ref="temp_unstr" domain_ref="regular_extract"/>

<domain id="unstructured"  n_glo="10000" type="unstructured" />

<domain id="regular_extract" ni_glo="360" nj_glo="180" type="rectilinear">
  <generate_rectilinear_domain/>
  <interpolate_domain/>
  <extract_domain ibegin="20" ni="50" jbegin="100" nj="60" />
</domain>
```

Hands-on 7-8



 temporal_splitting

- This filter generates a data flux which has one extra dimension comparing to the input flux
 - Ex : A 2D field over 4 time steps is transformed into a 3D field with the last dimension of size 4
- To use this filter, you need to at first reshape the input grid : add one scalar element to the grid to increase the grid's dimension

```
<grid id="grid_src">
  <domain domain_ref="domain"/>
</grid>
```



```
<grid id="grid_dst_1">
  <domain domain_ref="domain"/>
  <scalar id="scalar"/>
</grid>
```

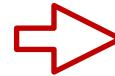
- Source field data can be passed on the new grid using arithmetic operation

```
<field id="field_src" grid_ref="grid_src" />

<field id="field_dst_1" grid_ref="grid_dst_1" > field_src </field>
```

- You can then use the temporal splitting filter as other spatial transform filters

```
<grid id="grid_dst_1">
  <domain domain_ref="domain"/>
  <scalar id="scalar"/>
</grid>
```



```
<grid id="grid_dst">
  <domain domain_ref="domain"/>
  <axis id="axis_ts" n_glo="4">
    <temporal_splitting />
  </axis>
</grid>
```

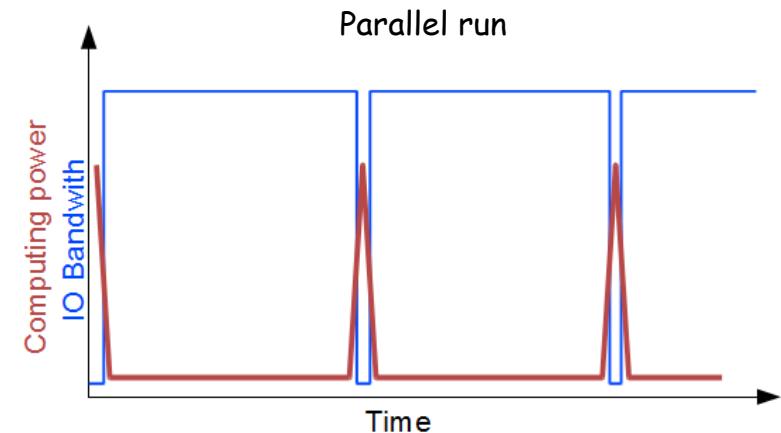
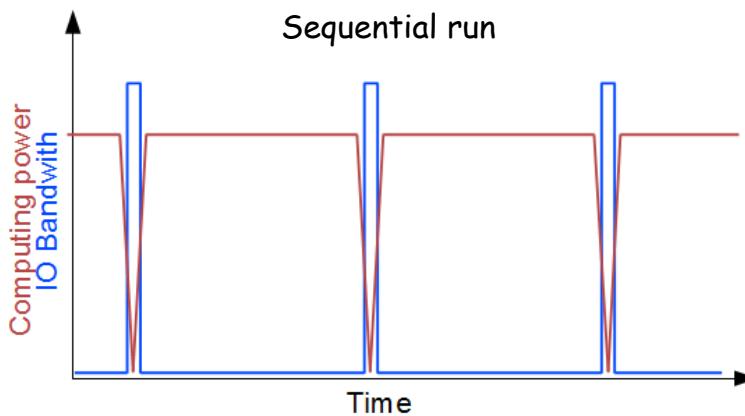
Hands-on 8

A good tool for visualize workflow

- Field attribute
 - ➡ (bool) **build_workflow_graph** : set to “true” to enable workflow
- Can be inherited by reference
- https://forge.ipsl.jussieu.fr/ioserver/chrome/site/XIOS_TEST_SUITE/graph.html
- Interactive
- One graph file per context.
 - ➡ `graph_data_*.json`
- Can be useful for debugging

Hands-on 9

Performance



- IO become a big bottleneck in parallel computing up to O(10000) cores
 - ➡ Often, data are gathered to one master process which write file
 - ➡ Ok if done just for initialization or finalize but memory problem may occur
 - ➡ Big impact on computing performance

- One file by process ?
 - ➡ Good way to achieve moderate scalability but :
 - ➡ Depending on the file system, performance may break down when attempting to write simultaneously thousand of files
 - ➡ Files need to be rebuilt into a single file in order to be analyzed
 - ➡ Rebuilt may take a longer time than the simulations

Using parallel IO ?

- ▶ Best way to achieve scalable IO without rebuild file
- ▶ But difficult to aggregate a lot of I/O bandwidth with a big number of writing processes
- ▶ Parallel IO are very less scalable than models due to hardware restriction (pricy and not took into account for performance evaluation)
- ▶ Impact on the computing performances.

Using asynchronous parallel IO ?

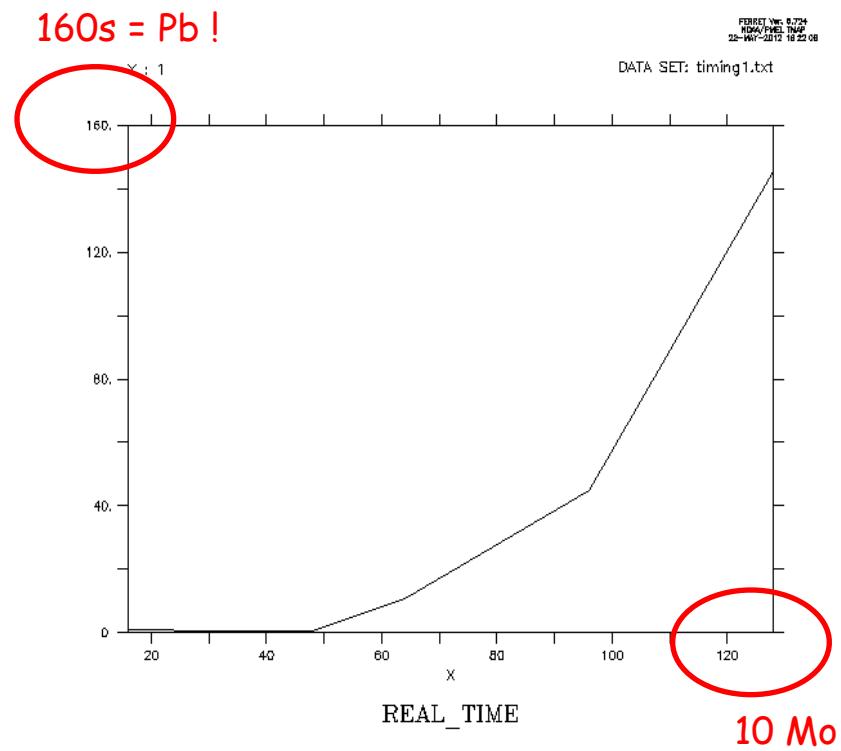
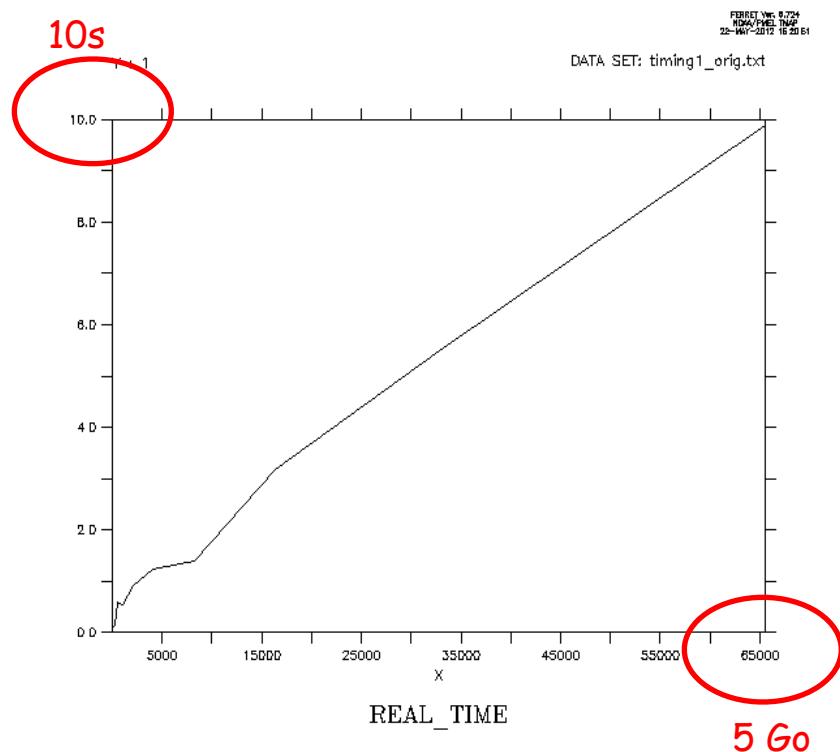
- ▶ Good way to overlap IO by computing
- ▶ MPI/IO : difficult to manage, load balancing problem...
- ▶ High level library (HDF5, netcdf...) generally don't implement asynchronous IO.

I/O performances are very system dependent

- ▶ Example : Curie Tier 0 computer with LUSTRE file system
- ▶ 150 GB/s theoretical capability
- ▶ Optimally tuned MPI-IO parallel benchmark : 10 GB/s
- ▶ HDF5 layer ~ 5GB/s
- ▶ NETCDF4-HDF5 layer ~ 4GB/s

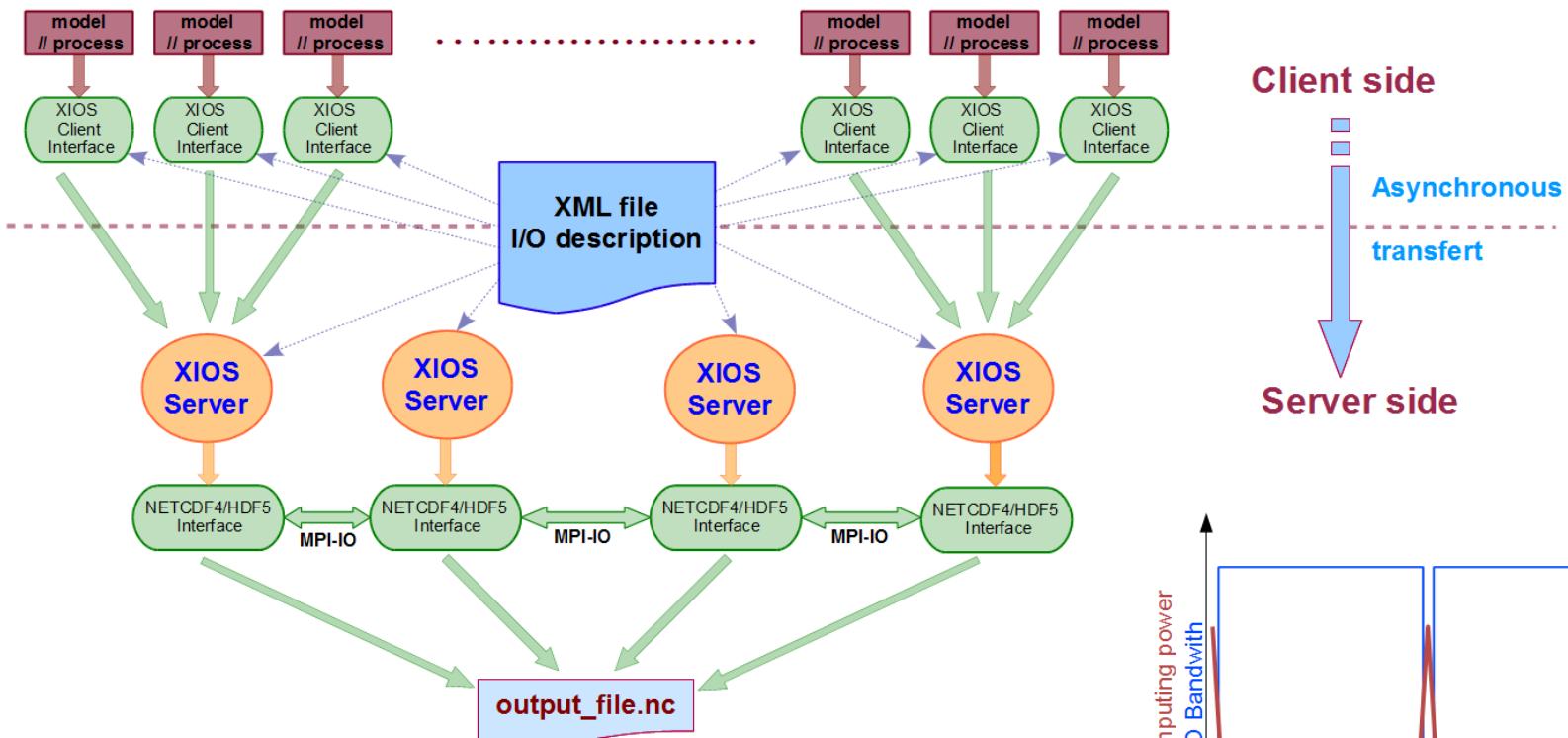


- Achieving good parallel IO performance is not so easy :
 - A lot of recipes to avoid very bad performance
 - Example with netcdf4, trying to perform naïve parallel IO

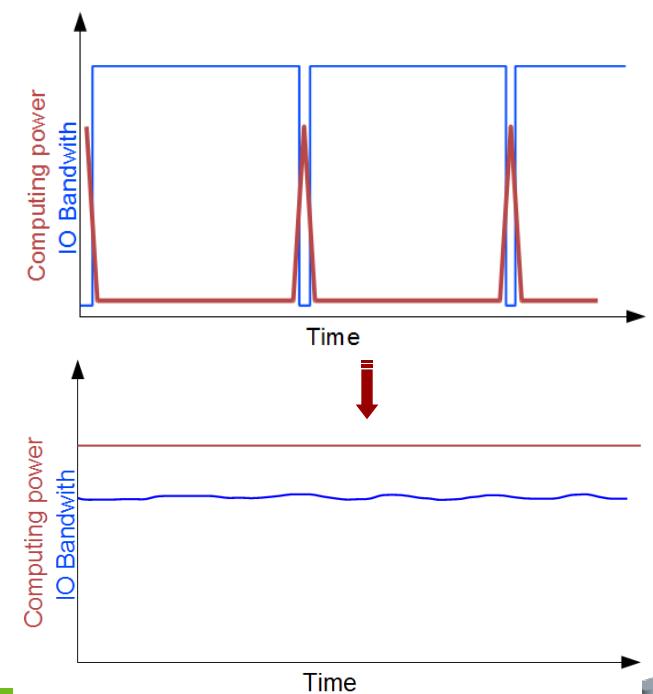
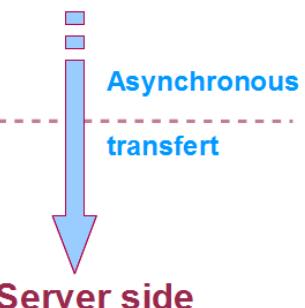


Multiple file on 16 CPUs : 1 file by process = 16 files

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



Client side



XIOS servers

Pool of process dedicated to parallel I/O

XIOS : a software Burst Buffer ?

- Data are written all along the simulation
- Smoothing I/O peaks
- Constant I/O flow to file system
- Overlap I/O by computation

Complex and fully asynchronous protocol

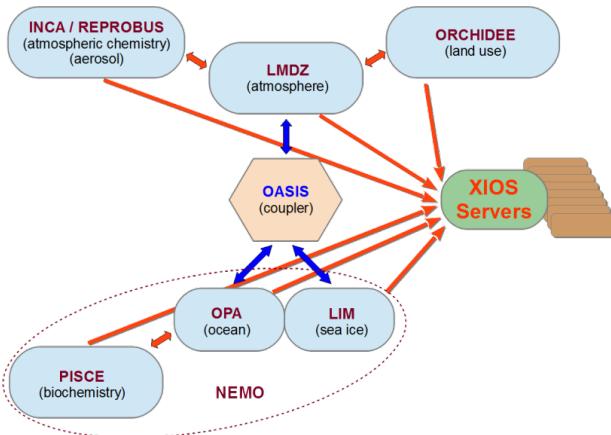
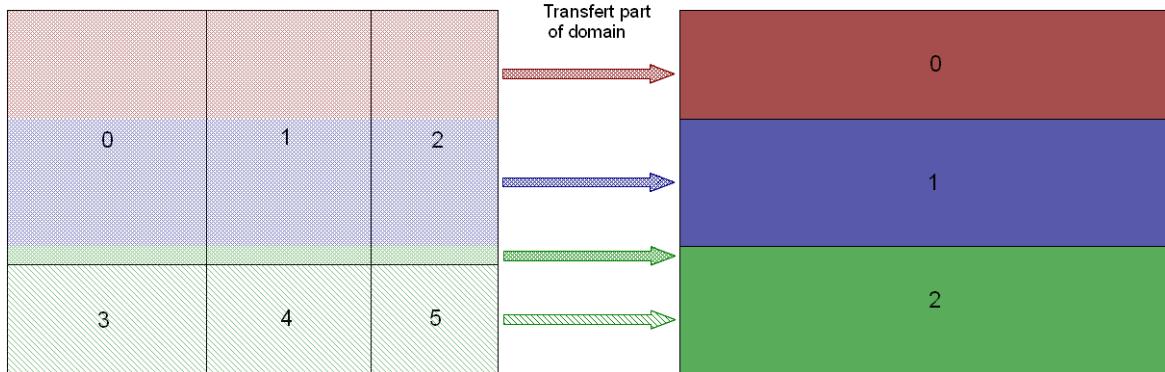
- One way to send data from clients to servers
 - One way to receive data from servers to clients



Different data distribution between client and servers

Client side

Server side



- 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

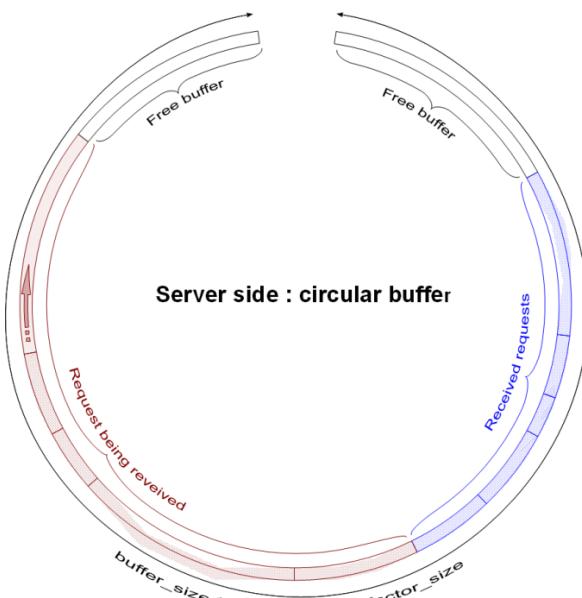
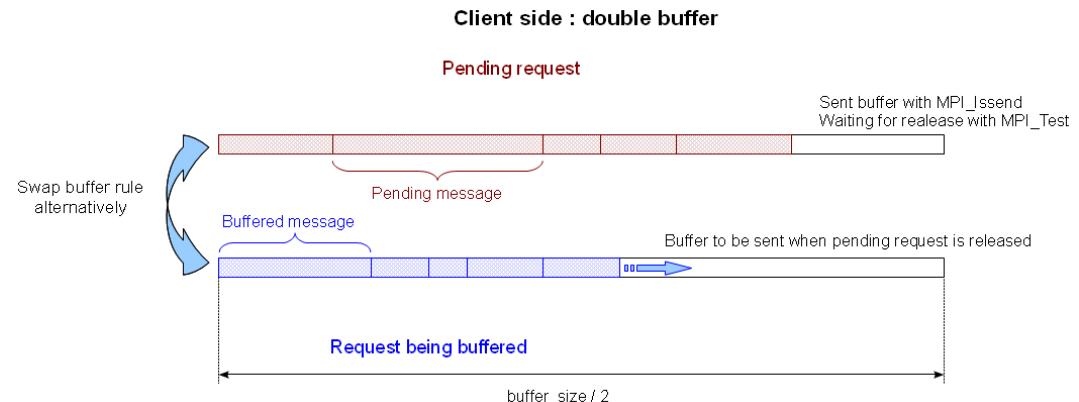


 Large usage of buffers

- Smoothing I/O peaks

 Client Side : double buffers

- Outgoing message in transfer
- Bufferization of the incoming flow



 Server Side : circular buffer

- Received request are processed
- In same time than receiving request from client

Overlapping data transfer and I/O by computing

Server mode

- MPMD mode
 - `mpirun -np 1024 model.exe : -np 16 xios_server.exe`

- Placing XIOS servers in parallel partition
 - Strongly hardware dependent
 - But generally better to spread servers on different computing nodes

Attached mode

- To make development easier XIOS provide an "attach" mode
 - Don't need to launch xios servers executable
 - `mpirun -np 12 model.exe`
 - XIOS act only as a library
- Each client is itself a server for other clients
 - Pool of servers is equal to the number of clients
- Synchronous
 - Client must wait for the data to be written before continue
- Each client make parallel write
 - performance issue



Why 2-level server?

- When number of XIOS servers increases, parallel I/O becomes inefficient due to I/O bandwidth
- Want XIOS servers to work with different output file

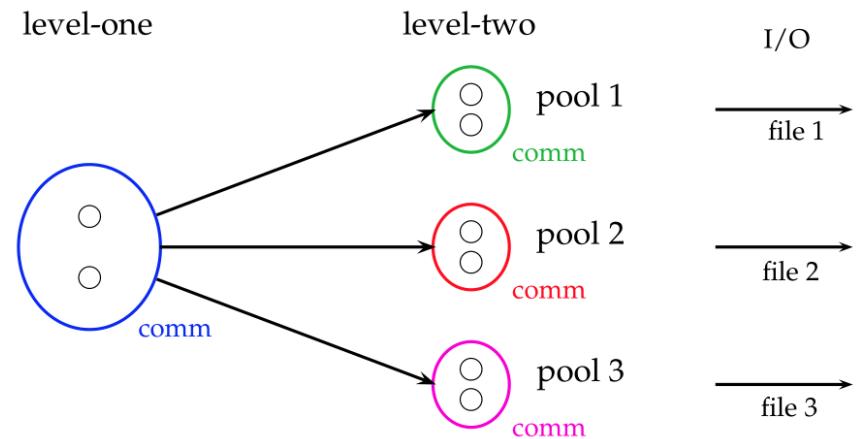
Intermediaries (level one) and writers (level two)

- Level-one servers will receive data from clients, redistribute, and send data to subsets of level-two servers (called "pools")
- Level-two servers will do the I/O
- Each file is written by only one pool
- No compression
- But if 1 process is assigned per pool (default option), I/O is then sequential and HDF5 compression can be used

Hands-on 10

Parameters: (context id="xios")

- (bool) **using_server2** : default **false**
- (integer) **ratio_server2** : default **50**
- (integer) **number_pools_server2** :
sets the number of server-two pools
(default is number of second level servers)



Performance report

- Report is generated at XIOS finalization

Client side : xios_client_00.out

```
-> report : Performance report : total time spent for XIOS : 32.3497 s
-> report : Performance report : time spent for waiting free buffer : 1.1336 s
-> report : Performance report : Ratio : 3.50421 %
-> report : Performance report : This ratio must be close to zero. Otherwise it may be useful to increase
buffer size or numbers of server
```

Server side : xios_server_00.out

```
-> report : Performance report : Time spent for XIOS : 51.0071
-> report : Performance report : Time spent in processing events : 21.5263
-> report : Performance report : Ratio : 42.2026%
```

Client side : Time spent for waiting free buffer is small compare to total time

- Every thing is OK, no impact of I/O on computing time

Client side : Time spent for waiting free buffer is not insignificant

- Server side : if ratio (total time / time for process event) is close to 100%
 - I/O throughput is not enough fast to maintains asynchronism
 - Add more servers
- Servers side : if ratio is much less than 100% (60-80%)
 - Servers are not overloaded but cannot absorb and smooth I/O peaks
 - Buffer are to small and need to be increased

Memory consumption

- XIOS consumes memory internally
 - XIOS uses large transfer buffer
 - Part of memory is consumed by NETCDF4/HDF5
 - But generally, memory consumption is scalable (client & server)
 - Information about memory usage
 - Buffer size is automatically computed
 - Can be different for each communication channel (client-server couple)
 - Dependent of the parallel data distribution
 - 2 buffers for each client-server couple
 - 1 for sending data from client to server (I/O write)
 - 1 for receiving data from server to client (I/O read)

Client side : xios client 00.out

-> report : Memory report : Context <atmosphere> : client side : total memory used for buffer 2932872 bytes
-> report : Memory report : Context <atmosphere> : server side : total memory used for buffer 209733 bytes
-> report : Memory report : Minimum buffer size required : 209730 bytes
-> report : Memory report : increasing it by a factor will increase performance, depending of the volume of data wrote in file at each time step of the file

Server side : xios server 00.out

-> report : Memory report : Context <atmosphere_server> : client side : total memory used for buffer 209733 bytes
-> report : Memory report : Context <atmosphere_server> : server side : total memory used for buffer 1710664 bytes



Managing buffer size

- Buffer sizes are automatically computed
- User can choose between 2 behaviors (parameter **optimal_buffer_size**) :
 - Buffer sizes optimized for memory
 - ➡ Size adjusted to the biggest transfer
 - ➡ Minimal memory consumption for buffer
 - ➡ But losing most part of asynchronous transfer
 - Buffer sizes optimized for performance
 - ➡ Sizes are adjusted to bufferize all data between two output period
 - ➡ Fully asynchronous
- User can adjust size by itself using a multiplying factor
 - ➡ (double) **buffer_size_factor**

Performance : what to expect...

+ XIOS is used on simulation with O(10 000) cores and more...

- Ex: CINES Big Challenges 2014 : DYNAMICO 1/8° and NEMO 1/60°

+ Bench test case : NEMO 1/12°

- Gyre configuration : $4322 \times 2882 \times 31$: 8160 cores
- Curie supercomputer : Lustre file system : theoretical Bandwidth : 150 GB/s (announced)
- Practical Bandwidth : NETCDF4/HDF5 file format : parallel write access on a single file (tuned): ~ 5 GB / s
- 6 days simulation (2880 time steps) ~ 300 s run s

+ 6-hours frequency output files (~200 GB of data produced, 4 files)

- 8160 NEMO, 32 XIOS servers
- +5% penalty for I/O (comparable to OS jittering)

+ Extreme test case : hourly output files (~1.1 TB of data produced, 4 files)

- 8160 NEMO, 128 XIOS servers (1.5 % resources for I/O)
- 15-20% penalty for I/O
- 3.6 GB/s I/O flux continuously
- **Generated data amount : ~300 TB by day, ~10 PB by month**



XIOS context is used for parametrization

- Specific XIOS context in XML file
- Used only for reading variable value
- Actually, all parameters are optional, just override default value

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

    <variable id="optimal_buffer_size" type="string">performance</variable>
    <variable id="buffer_size_factor" type="double">1.0</variable>
    <variable id="min_buffer_size" type="int">100000</variable>
    <variable id="using_server" type="bool">false</variable>
    <variable id="using_oasis" type="bool">false</variable>
    <variable id="info_level" type="int">50</variable>
    <variable id="print_file" type="bool">true</variable>

  </variable_definition>
</context>
```



- (string) **optimal_buffer_size** : specify buffer sizing behavior (**default : "performance"**)
 - ➡ "performance" or "memory"
- (double) **buffer_size_factor** : multiplying the computed buffer size by this factor
 - ➡ Use with caution
- (integer) **min_buffer_size** : fix the minimum size of buffers
 - ➡ Use only in case of bad computed size
 - ➡ Can help to workaround an unexpected problem
- (boolean) **using_server**: specify "server mode" or "attached mode"
 - ➡ XIOS try to determine itself the chosen mode by analyzing MPI communicator
 - ➡ Useful only for coupled model configuration
- (boolean) **using_oasis** : used when interfaced with oasis (expert mode), (**default=false**)
- (integer) **info_level**: level of xios information output (**0-100**), **0** nothing, **100** full, (**default=0**)
- (boolean) **print_file** : if true, xios standard output and error are redirected in files indexed by process rank, (**default=false**)

Hands-on 11

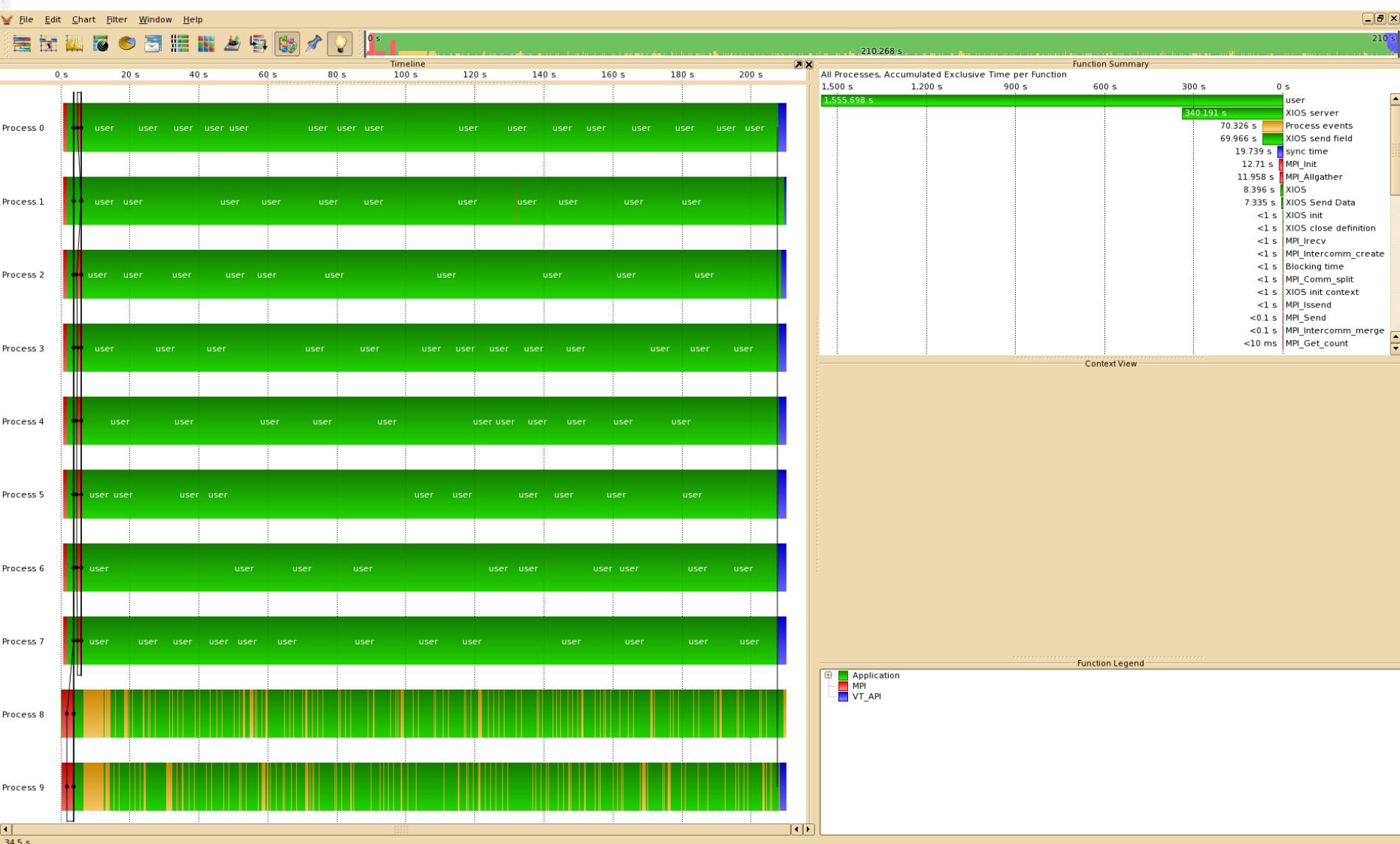
Understand and analyze XIOS servers performance

- Build a toy model
- Field is sent and written at each time step
- Some extra working time is simulate by a waiting call

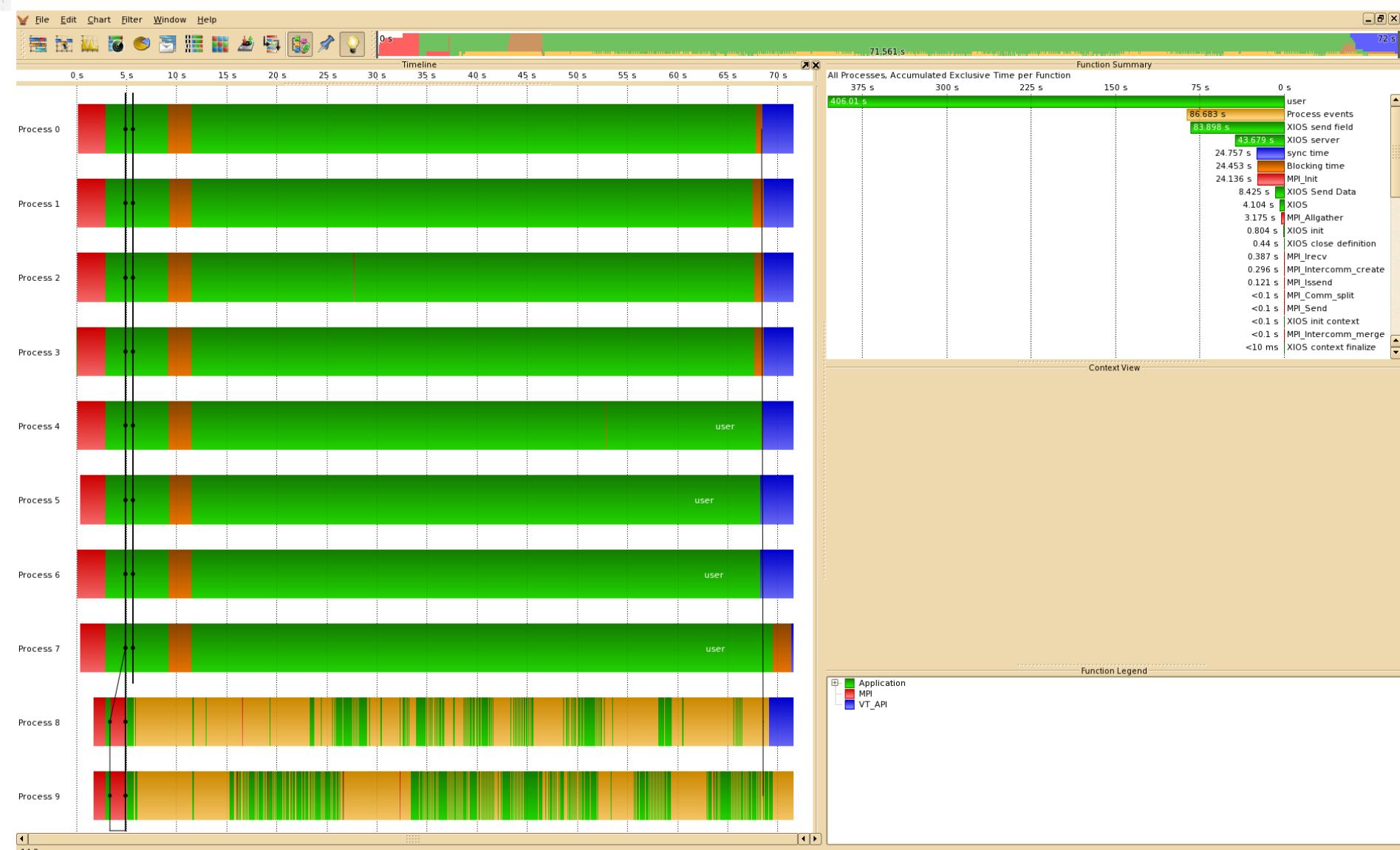
```
! Entering time loop
DO ts=1, 1000
  CALL xios_update_calendar(ts)
  CALL xios_send_field("field", field)
  CALL wait_us(80000) ! Wait 80 milliseconds to simulate some works
ENDDO
```

- Look at parallel vampir trace
 - Green : application time
 - Red : MPI function time
 - Orange : server working time
 - Brown : client waiting for free buffer and blocking
- Make experiments by decreasing working time compared to I/O output

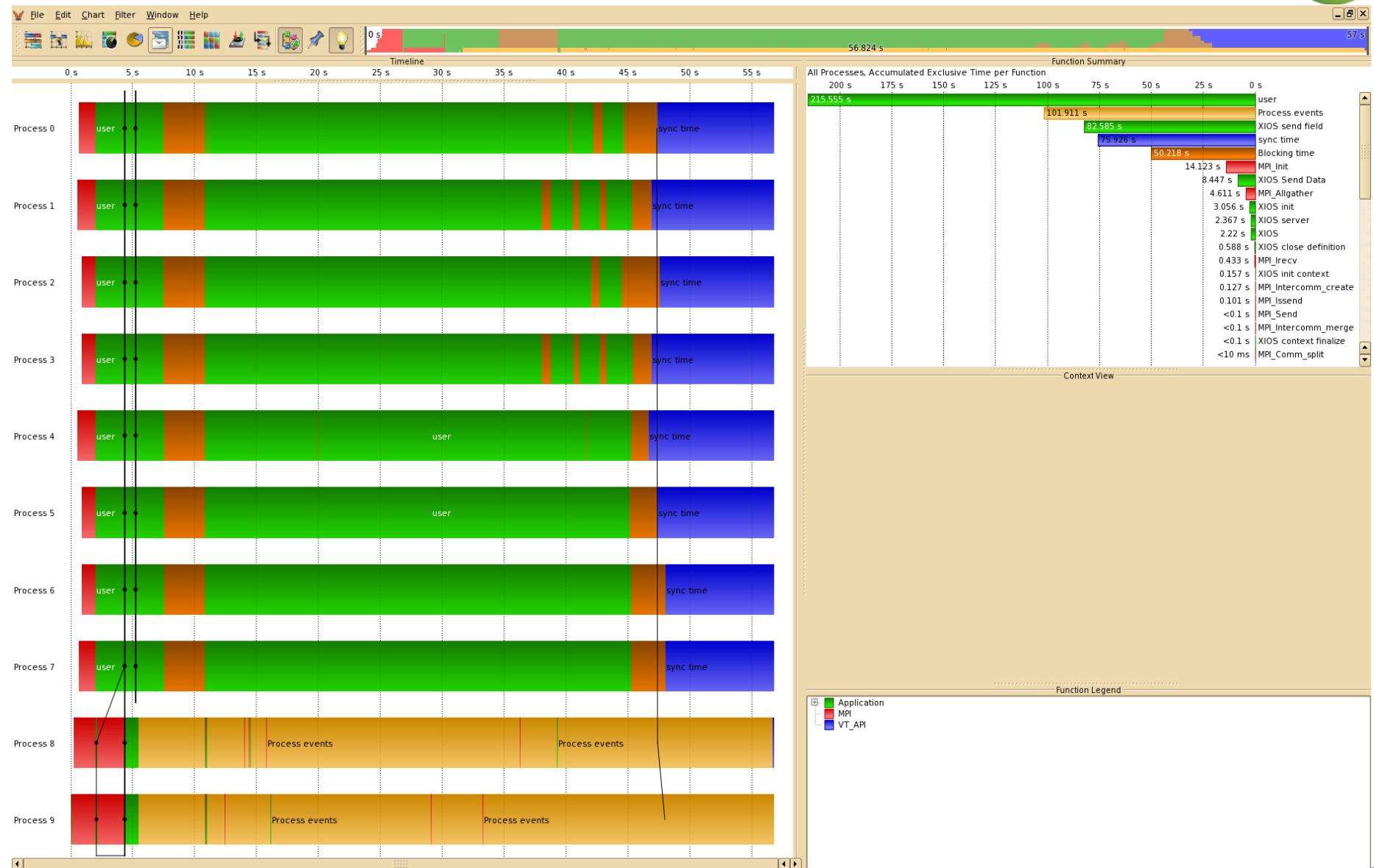
8 clients – 2 servers : working time by iteration: 80 ms



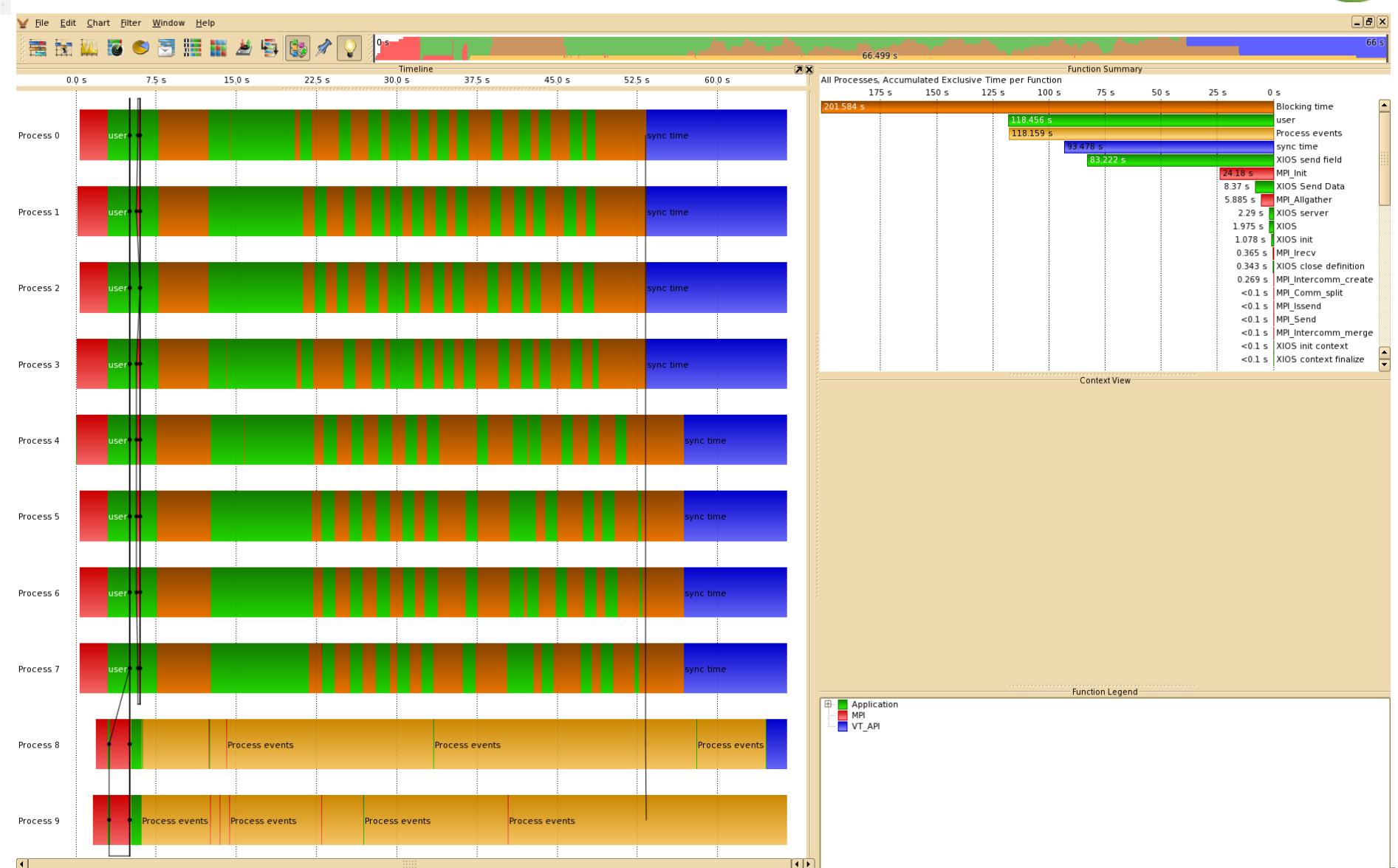
8 clients – 2 servers : working time by iteration: 20 ms



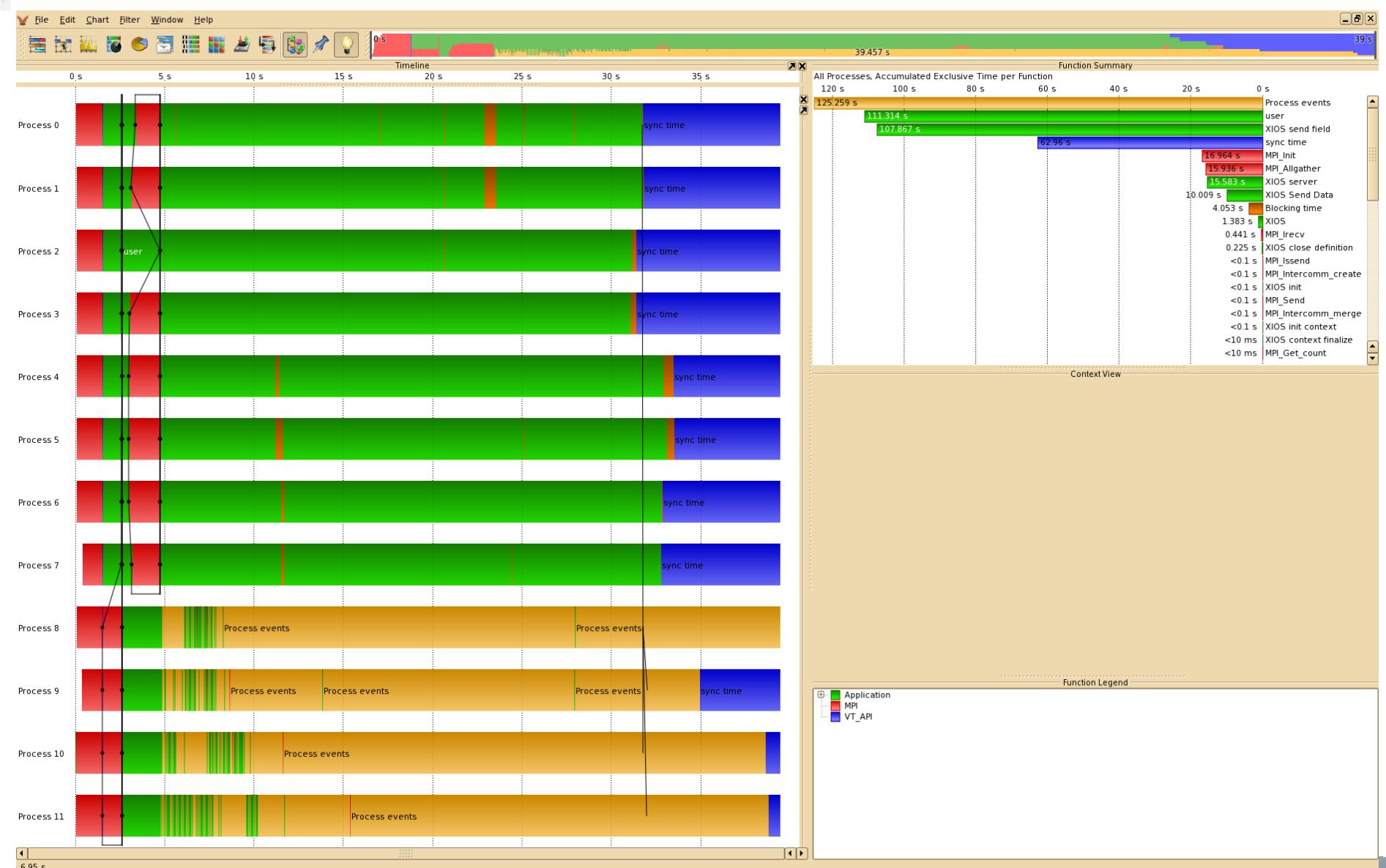
8 clients – 2 servers : working time by iteration: 10 ms



8 clients – 2 servers : working time by iteration: 5 ms



8 clients – 4 servers : working time by iteration: 5 ms



8 clients – 8 servers : working time by iteration: 5 ms

