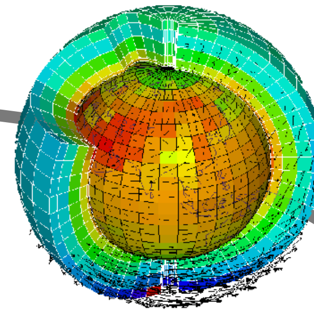
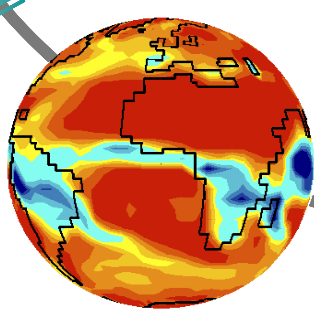
Yann Meurdesoif, A. Caubel, R. Lacroix, J. Dérouillat, M.H Nguyen

$$\frac{\partial \mathbf{U}}{\partial t} + 2\Omega \times \mathbf{U} = .$$

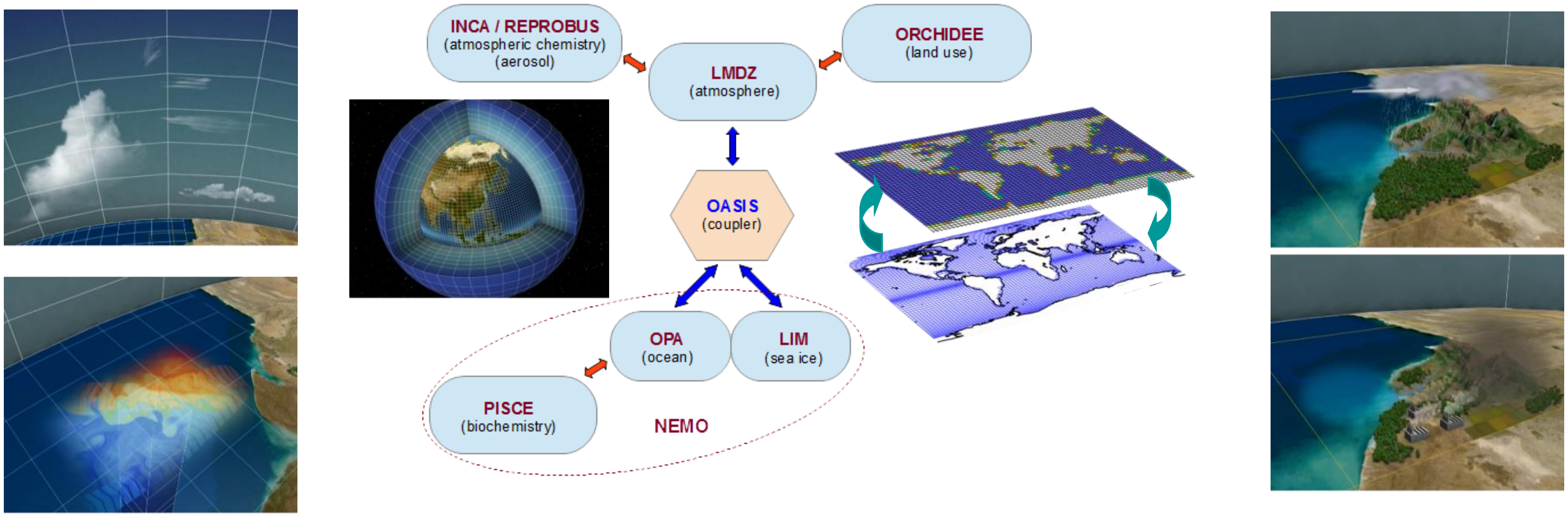


# XIOS TUTORIAL

```
.CTER*(*) pfilename
..EGER,INTENT(IN) :: pim,pjm
EAL,DIMENSION(pim,pjm),INTENT(IN) :.
INTEGER,INTENT(IN):: par_orix, par_sz,
INTEGER,INTENT(IN) :: pitau0
REAL,INTENT(IN) :: pdate0, pdeltat
INTEGER,INTENT(OUT) :: pfileid, phorii

INCLUDE "netcdf.inc"

INTEGER :: ncid, iret
TEGER :: leng, lengf, lenga
CTER(LEN=120) :: file, tfil
ER*30 :: timenow
```



## Context : Earth system models




- **Complex coupled model, long simulations, a lot of data generated…**

- **Example for past CMIP5 exercise**
  - 167 numerical experiments, 25000 simulated years (20th and 21th century, last millennium, paleo-climate).
  - Monthly, daily and high frequency (6h) files
    - Up to 800 variables for each files
    - A lot of metadata associated to each variable
  - 5 Millions generated files
  - 2 PB of data

# DATA PRODUCTION IN CLIMATE MODELING

- **CMIP6 next exercise**
  - Beginning mid-2016
  - x 7 => 14 PB of data to be generated, stored and distributed (ESGF grid)

**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
  - Parallel I/O efficiency ?
- **Complexity and efficiency of post-treatment chain to be suitable for distribution and analysis**
  - Files rebuild, time series, seasonal means…
  - Mesh regridding, interpolation, compression…
  - Resiliency ?

**More time is spent to output and post-treat data than running the simulation !**

LSCE

## XIOS is addressing all these challenges

### Flexibility in management of I/O and data definition

- Using an external XML file parsed at runtime

### I/O Performance issue

- Dedicated Parallel and Asynchronous I/O server

### Post-treatment issue

- Integrate internal parallel workflow and dataflow
- Post-tretment can be performed "In Situ"

# XIOS - INTRODUCTION

## XIOS is a ~5 years old software development

### XIOS 2.0 : ~ 85 000 code lines, written in C++, interfaced with Fortran models

- Open Source CECILL Licence
- Code versioning : SVN (subversion)
  - `XIOS 2.0 (dev) : forge.ipsl.jussieu.fr/ioserver/svn/XIOS/trunk`
  - `XIOS 1.0       : forge.ipsl.jussieu.fr/ioserver/svn/XIOS/branchs/xios-1.0`

### Used by an increasing variety of models

- IPSL models : NEMO, LMDZ, ORCHIDEE, INCA, DYNAMICO
- LGGE (MAR), Ifremer (ROMS, MARS3D)
- European NEMO consortium
- MétéoFrance / CNRM (ongoing) : Gelato, Surfec, Arpège climat (CMIP6 production)
- European models (in evaluation) : MetOffice (Hadgem ? , MONC, Gung-Ho?), ECMWF (Open IFS ?, EC-EARTH ?)

# XIOS - INTRODUCTION

## 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)
- Manh Ha Nguyen (LSCE)
- Remy Lacroix (ex-LSCE, CNRS/IDRIS)
- Julien Dérouillat (CEA/MdS)

# XIOS Phylosophy

## Philosophy:

- **Each time step, models expose part of their data through a minimalist interface**
  - Identifier (ASCII string) + address (pointer) of the data
    - Output:
    - Input:

```
CALL xios_send_field("field_id",field_out)
CALL xios_recv_field("field_id",field_in)
```

- **External XML File :**
  - Describe the incoming data flow from models using XML attributes
  - Describe the workflow applied to the incoming data flow
  - Describe the data flow end point => 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**
  - All XML definitions can be completed or created from model

# GETTING STARTED : "Hello World" test case

## Output a variable in a "hello_word" file like an averaging at 1 day frequency

- Define 1D-Axis and horizontal 2D-domain associated with the model variables

```
<domain id="horizontal_domain" ni_glo="100" nj_glo="100"/>
<axis   id="vertical_axis" n_glo="100" />
```

- Define grid associated with the  associated axis and domains
  - Example 3D grid :  100x100x100

```
<grid id="grid_3d">
  <domain domain_ref="horizontal_domain"/>
  <axis axis_ref="vertical_axis"/>
</grid>
```

- Define fields (variables)  associated with the grid

```
<field id="a_field"  grid_ref="grid_3d">
```

- Associate fields with files

```
<file name="hello_word"  output_freq="1day">
  <field field_ref="a_field" operation="average">
</file >
```

# GETTING STARTED : "Hello World" XML file

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

    <axis_definition>
      <axis id="vertical_axis" n_glo="100" />
    </axis_definition>

    <domain_definition>
      <domain id="horizontal_domain" ni_glo="100" nj_glo="100" />
    </domain_definition>

    <grid_definition>
      <grid id="grid_3d">
       < domain domain_ref="horizontal_domain" >
       < axis   axis_ref="vertical_axis" >
    </grid_definition>

    <field_definition >
      <field id="a_field" operation="average" grid_ref="grid_3d" />
    </field_definition>

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

  </context>
</xios>
```

# GETTING STARTED : "Hello World" Fortran side

```fortran
SUBROUTINE client(rank,size)
  USE xios
  IMPLICIT NONE
  INTEGER :: rank, size, ts
  TYPE(xios_time)      :: dtime
  DOUBLE PRECISION,ALLOCATABLE :: lon(:,:),lat(:,:),a_field (:,:)
  ! other variable declaration and initialisation .....

! XIOS initialization
  CALL xios_initialize("client", return_comm=comm)
  CALL xios_context_initialize("hello_word",comm)
! Complete horizontal domain definition
  CALL xios_set_domain_attr("horizontal_domain",ibegin=ibegin,ni=ni,jbegin=jbegin, nj=nj)
  CALL xios_set_domain_attr("horizontal_domain ",lonvalue_2d=lon, latvalue_2d=lat)
! Setting time step
  dtime%second=3600
  CALL xios_set_timestep(dtime)
! Closing definition
  CALL xios_close_context_definition()
! Entering time loop
  DO ts=1,96
    CALL xios_update_calendar(ts)
    CALL xios_send_field("a_field",a_field)
  ENDDO
! XIOS finalization
  CALL xios_context_finalize()
  CALL xios_finalize()
END SUBROUTINE client
```

# XML TERMINOLOGY

## XML : Extensible Markup Language

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

## Tag : a markup construct that's begin by "<" and ends by ">"

- Start-tag : `<field>`
- End-tag : `</field>`

## Element : construct delimited by a start-tag and an end-tag

- May be empty : `<field></field>`
  - Can be written with empty-tag syntax : `<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>
```

  - Only use in XIOS to define arithmetic's operations

# XML TERMINOLOGY

- **Attributes : a construct composed of a pair : name="value" defined into a start-tag or an empty tag element**
  - Ex : Element field has 3 attributes : **id**, **name** and **unit**
  - <field id="temp" name="temperature" unit="K" />

- **Comments : delimited by starting tag comment <!-- and ending tag comment -->**
  - <field> *<!-- this is a comment, not a child nor a content -->* </field>

- **XML document must be well-formed**
  - XML document must contains 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

# GENERAL XIOS-XML SYNTAX

## XML master file must be `iodef.xml`

- Parsed first at XIOS initialization
- Root element name is free
- Root element can only contain <**context**> type elements.

## 5 main element families: represent objects type stored into XIOS database

- **context** : isolate and confine models definition, avoiding interference between them
- **field** : define incoming field from model
- **axis** : define 1D-axis
- **domain** : define 2D-domain
- **grid** : define multi-dimensional grid, composed of axis and domains
- **file** : define input or output file
- **variable** : define parameters for models or for XIOS parameterization

## These families can be declined into 3 flavors (except for context elements)

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

# GENERAL XIOS-XML SYNTAX

- **Each element may have several attributes**
  - ie : `<file id="out" name="output" output_freq="1d" />`
  - Attributes give information for the related element
  - Some attributes are mandatory, so error is generated without assigned value (small part)
  - Some other are optional but have a default value
  - Some other are completely optional

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

# GENERAL XIOS-XML SYNTAX

## Special attribute `id` : identifier of the element

- Used to take a reference to the corresponding element
- Is unique for a kind of element
  - Same id for different elements refers internally to the same object
  - Be very careful when reusing same ids, not advised
  - Root elements are equivalent to group elements with a fixed id
  - Ex: `<field_definition ….>` ⇔ `<field_group id="field_definition" …>`
- Is optional, but no reference to the corresponding element can be done later

## 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
  - attribute value give the name of the file to be inserted in the database

```
File iodef.xml :

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

file nemo_def.xml :

<context id="nemo" >
  <field_definition … >
...
</context>
```

# INHERITANCE MECHANISMS

## Why Inheritance ?

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

## Inheritance by grouping : parent-children 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" />
      <field id="woce_eff"   long_name="effective ocean vertical velocity"  unit="m/s" />
   </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="average" enabled="true" />
```

# INHERITANCE MECHANISMS

## Inheritance by reference

- Only for field, domain and axis elements
  - **field** => **field_ref** attribute
  - **domain** => **domain_ref** attribute
  - **axis** => **axis_ref** attribute

- Source element inherit all attributes of referenced element
  - Attributes already defined in source element are not inherited

```
<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
  - Be careful with potential side effects

- Reference may have additional meaning
  - Ex : **field_ref** bind also the data value of the referenced field
  - See later in tutorial matching section

## Why Context ?

- Context is similar to a "namespace"
- Context are isolated from each other, no interference is possible
  - Ids can be reused with no side effects between context
- For parallelism, each context is associated with its own MPI communicator
  - No interference between MPI communication
- Generally a context is associated to one model
  - Principle of modularity
- A model can declare more than one context

## Context element : <context/>

- Can appear only as a child element of the root XML element
- Must have an id
- Can contains only calendar or other element root 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 />
```

# CALENDAR

## Each context must define an associate calendar

- One calendar by context
- Define a calendar type
  - Date and duration operation are defined with respect to the calendar
- 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
- `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

# CALENDAR

## 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:min:sec`**
  - Ex. : "`2015-12-15 10:15:00`"
- Partial definition are allowed taking taking into account rightmost part
  - Ex. "`2015-12`" equivalent to "`2015-12-01 00:00:00`"

# CALENDAR

- Date can be also define with a duration offset
  - Useful for defining a calendar based on standard units (seconds for example)
  - Ex. : "+3600s"
  - Or mixt : "2012-15 +3600s" equivalent to "2012-15-1 01:00:00"

## Specific attribute calendar

- type : define the calendar type
  - "Gregorian", "D360", "NoLeap" , "AllLeap", "Julian" or "user_defined"
- (date) time_origin : define the simulation starting date (fixed at "0000-01-01" by default)
- (date) start_date : define the starting date of the run (fixed at "0000-01-01" by default)
- (duration) timestep : 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="2015-12" timestep="1h"/>
   …
</context />
```

## Defining an user define calendar

- Planetology or paleo-climate can not use standard calendar

- Personalized calendar
  - Defining **day_length** in second (default 86400)
  - Defining **month_lengths** : number of days for each 12 months
  - Or if you don't want to specify month : **year_lenght** in second : months would be equally distributed

```
<calendar  type="user_defined" day_length="86400"
           month_lengths="(1,12) [31 28 31 30 31 30 31 31 30 31 30 31]" />
```

- Possibility to define leap year
  - Attributes : **leap_year_month**, **leap_year_drift**, **leap_year_drift_offset**
  - See documentation…

# Managing calendar, date and duration from Fortran interface

## Duration

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

```
TYPE(xios_duration) :: duration
duration%second = 3600.
Duration = 3600 * xios_second
```

## Date

- Fortran derived type : TYPE(xios_date)
  - Component (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 operation: duration±duration, duration*real, -duration, comparison : ==, /=
- Date operation : date - date, boolean operators : ==, >=, >, <=, <, /=
- Date and duration operation : date+duration, 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

# CALENDAR – FORTRAN INTERFACE

## Setting calendar from Fortran interface

- Within single call

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

- Or with individual call

```
SUBROUTINE xios_set_timestep(timestep)

SUBROUTINE xios_set_time_origin(time_origin)

SUBROUTINE xios_set_start_date(start_date)
```

# GRID DEFINITION

## Field geometry is provided by the underlying mesh description

- Describe field dimensionality (1D, 2D, 3D or more)
- Describe field topology (rectilinear, curvilinear, unstructured)
- Describe field data distribution for parallelism

## Describing the mesh : the grid element : <grid />

- Can describe element of any dimension : 0D (scalar), 1D, 2D, 3D, or more.
- Defined by composition of 1D-axis and 2D-horizontal domain
  - **axis** and **domain** elements
- Empty grid is representing a scalar

```
<grid_definition />
   <grid id="grid_3d">   <!-- grid 3D of dimension 100x50x20 -->
      <axis n_glo="100"/>
      <axis n_glo="50"/>
      <axis n_glo="20">
   </grid >

   <grid id="grid_4d">   <!-- grid 4D of dimension 100x100x50x20 -->
      <domain ni_glo="100" nj_glo="100" type="rectilinear"/>
      <axis n_glo="50"/>
      <axis n_glo="20">
   </grid >
</ grid_definition />
```

# 1-D AXIS

## Axis description : the axis element <axis />

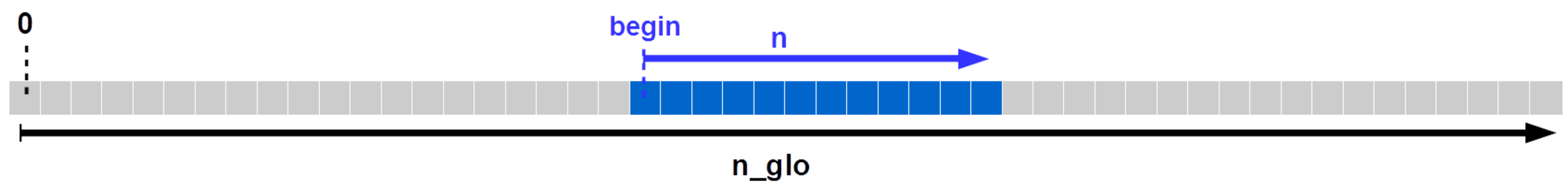
- Describe 1D axis, generally vertical axis

## Defining the global size of the axis

- (integer) `n_glo` attribute

## Defining the data parallelism distribution across MPI processes

- (integer) `n` attribute
  - Local axis size distribution
- (integer) `begin` attribute
  - Local axis distribution beginning with respect to the global axis
  - Follow C-convention, starting from 0.
- If nothing specified, the axis is considered as not distributed.




## Defining axis coordinate values and boundaries

- (real 1D-array) attribute : `value[n]`
- (real 2D-array) attribute : `bounds[2,n]`

# 1-D AXIS

## Defining how data are stored in memory

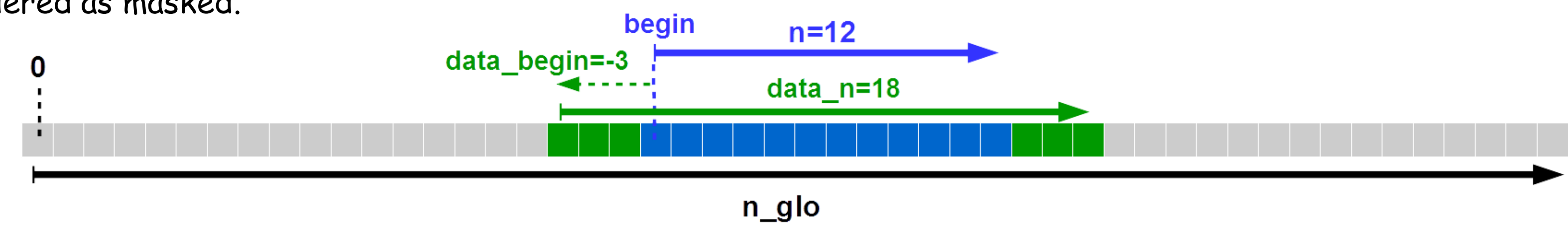
- Data are stored in memory as Fortran array
- But data can be masked, or ghost cells are not valid data, or axis value can be compressed
- XIOS will extract only required value from memory
- Must describe valid data with attributes
- Default is : whole data are valid

## Masking Data (optional)

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

## Defining ghost cells (optional)

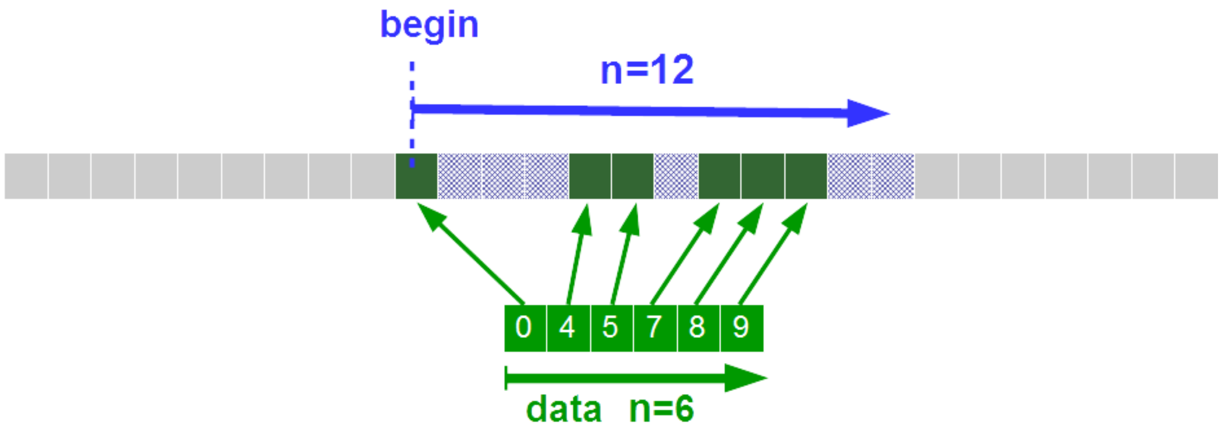
- (integer) **data_n** attribute
  - Size of the data in memory (default : **data_n**=**n**)
- Integer attribute : **data_begin**
  - Offset with respect to local axis distribution beginning (default : **data_begin**=0)
  - Negative offset : data outside of the local distribution will not be extracted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



# 1-D AXIS

## Defining compressed data

- Data can be compressed in memory (ex : land point), but may be decompressed to be output
- Undefined data are considered as masked and will be output as missing value
- (integer 1D-array) **data_index** attribute : data_index[data_n]
  - 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 attributes

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

# 1-D AXIS

## Using distributed axis within grid

- Global 3D-grid of size 100x50x20
- Describe a local 3D distribution of size 10x5x20 beginning at the index (20,10,0) of the global grid

```
<grid id="grid_3d">    <!-- grid 3D of global dimension 100x50x20 -->
   <axis n_glo="100" begin="20" n="10" />
   <axis n_glo="50"  begin="10" n="5" />
   <axis n_glo="20">  <!-- not distributed -->
</grid >
```

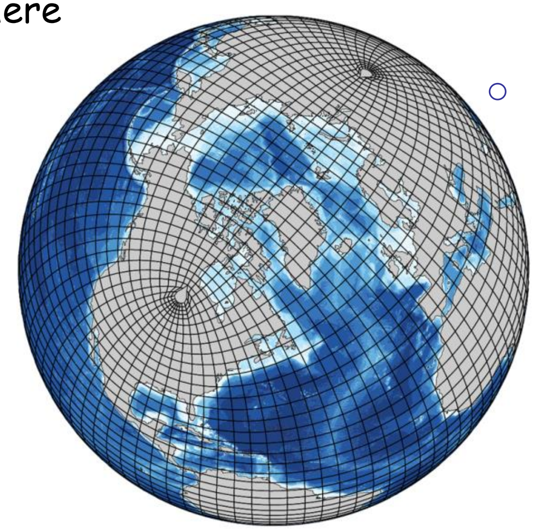
- Data distribution is different for each MPI process, not suitable for XML description
  - Attributes only known at run-time can be passed dynamically using the Fortran interface
  - See section Fortran interface setting attributes

## Masking grid point individually

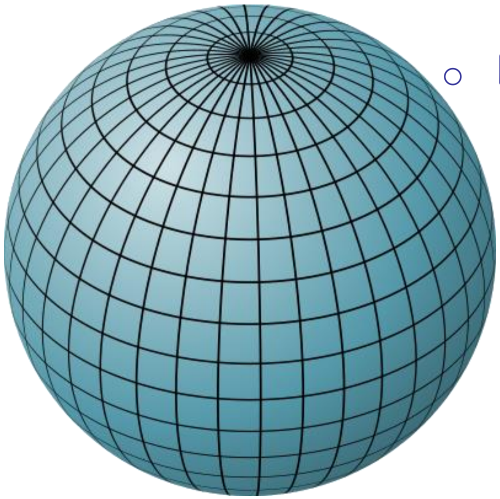
- In the last example, masking one point in the 3rd axis means masking a full 2D layer in the 3d grid
- Grid point can be masked using the mask attribute
- Regarding of the dimensionality of mask arrays, version mask_1d to mask_7d are allowed
  - Total mask size must be equal to the local domain size
  - Ex : `<grid id="grid_3d"  mask_3d="(10,5,20)[…,…, …]">`
  - or `<grid id="grid_3d"  mask_1d="(1000)[…,…, …]">`

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

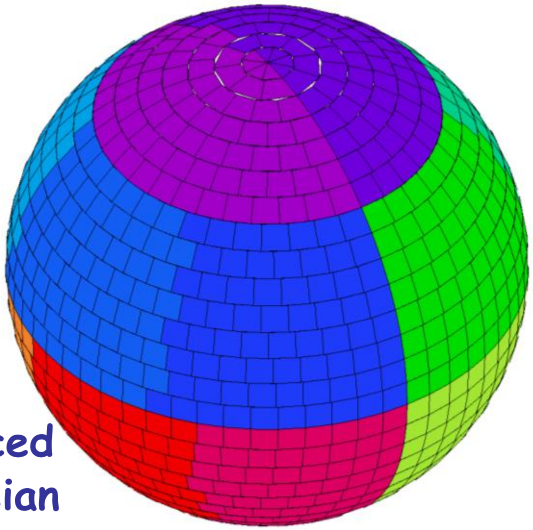
- Describe generally 2D layers mapping the surface of the sphere
- Large variety of 2D domains can be described
- `(string)` **type** attribute :
  - "`rectilinear`", "`curvilinear`" or "`unstructured`"



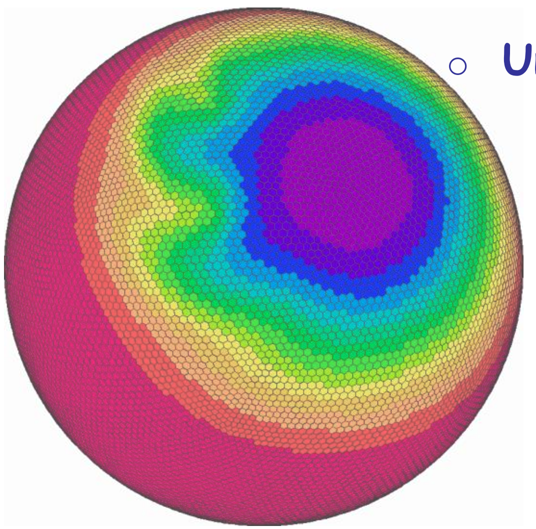
- Curvilinear



- Regular Cartesian (longitude x latitude)



- Reduced Gaussian

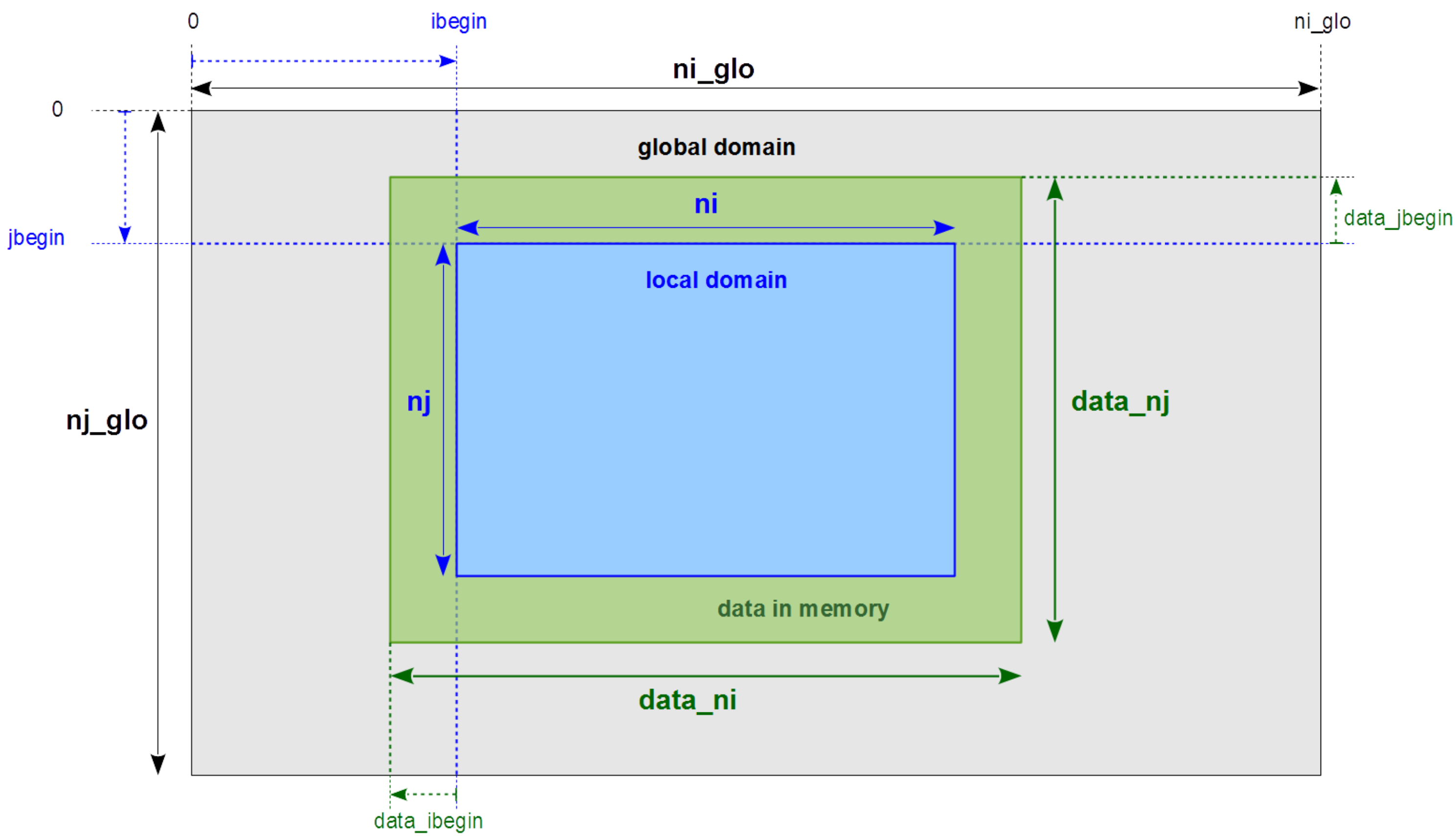


- Unstructured

# 2D-HORIZONTAL DOMAINS

## Rectilinear or curvilinear domains have a 2D description

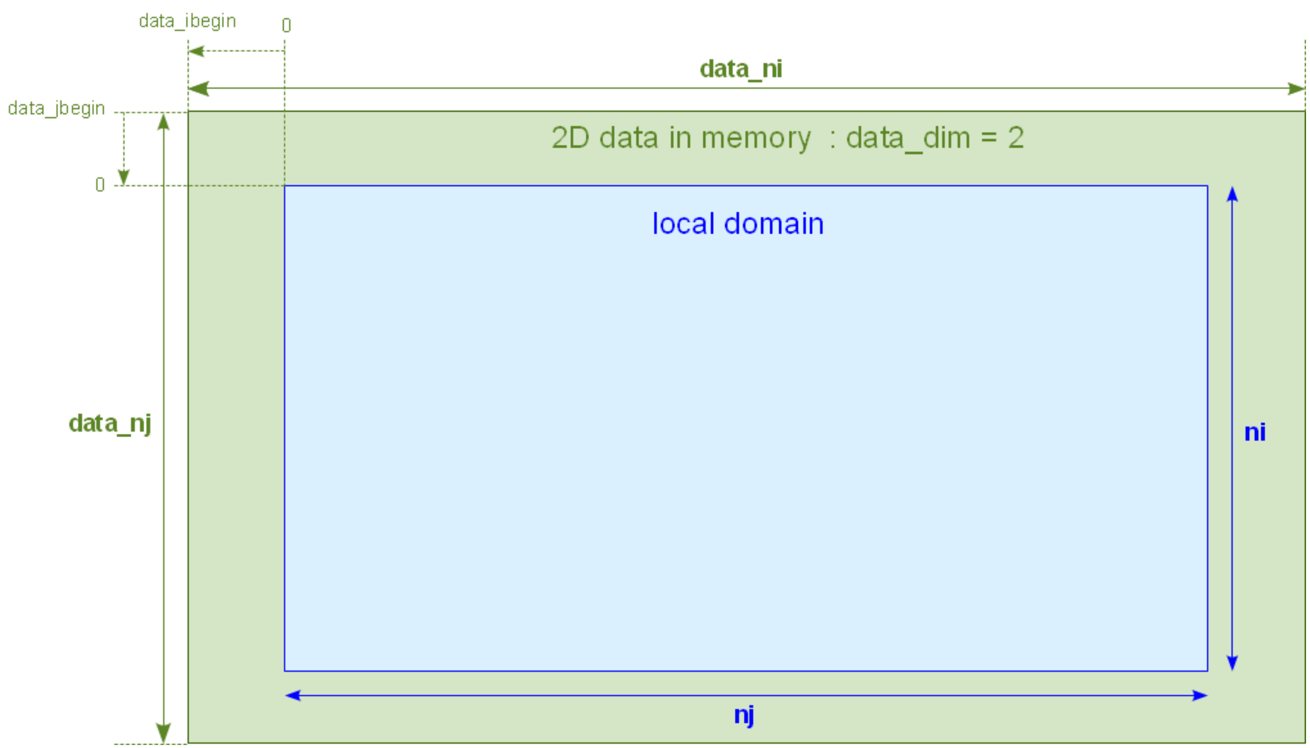
- **(integer)** **ni_glo, nj_glo** : global domain size for each direction (longitude and latitude for rectilinear)
- **(integer)** **ibegin, ni, jbegin, nj** : local domain definition



# 2D-HORIZONTAL DOMAINS

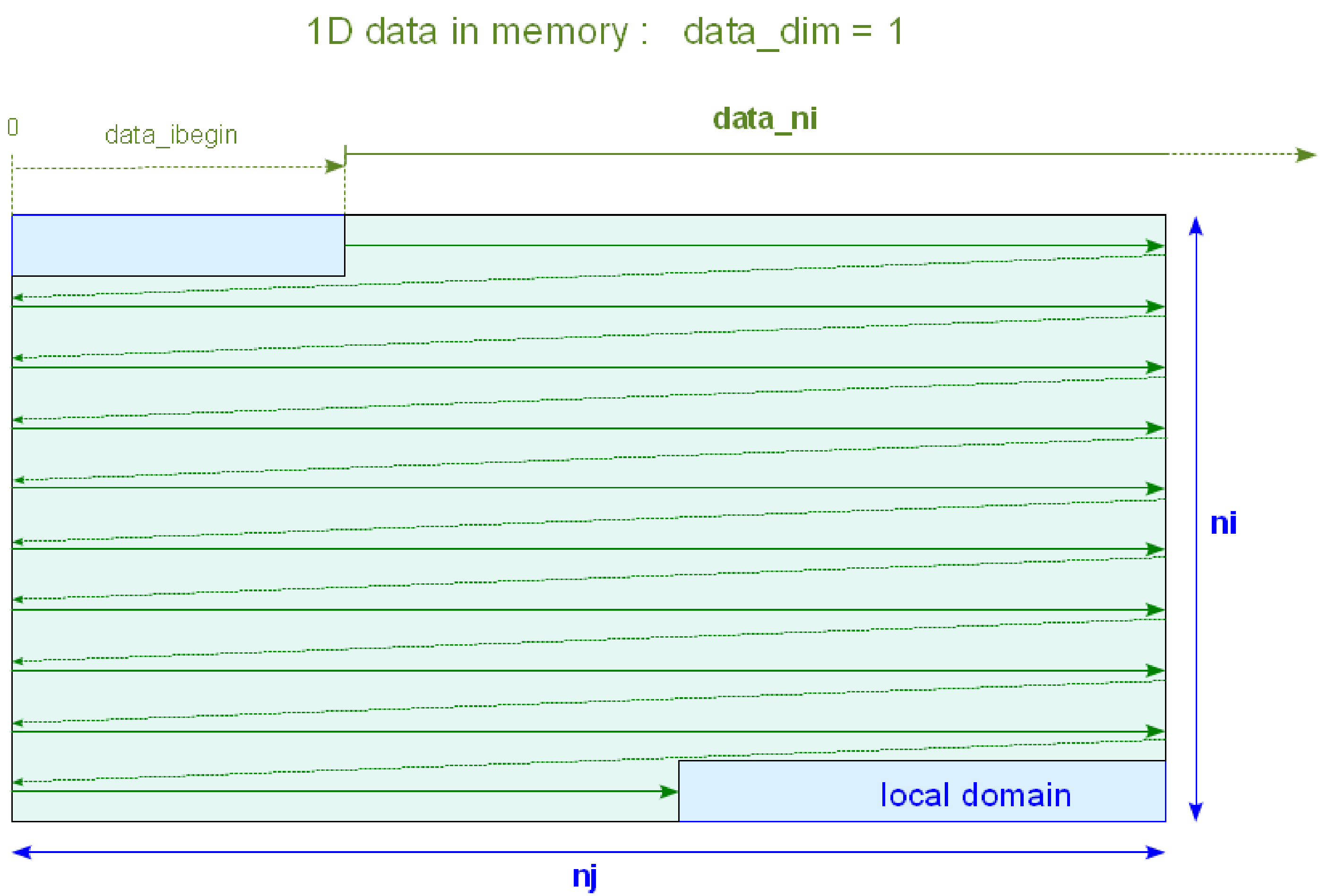
## Data representation in memory : similar to 1D-axis but for 2 dimensions

- Can be 1D-array (horizontal layer as a vector) or 2D-array
  - `(integer)` `data_dim` attribute : 1 or 2
- `(integer)` `data_ni` : size of the first array dimension (default : `data_ni=ni`)
- `(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



# 2D-HORIZONTAL DOMAINS

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



# 2D-HORIZONTAL DOMAINS

## Unstructured domain have a 1D description

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

## Compressed data (i.e land use)

- 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

- (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

# 2D-HORIZONTAL DOMAINS

## Defining coordinates

- For rectilinear domain
  - `latvalue_1d[ni]` : latitude coordinates
  - `lonvalue_1d[nj]` : longitude coordinates

- For curvilinear
  - `latvalue_2d[ni,nj]` : latitude coordinates
  - `lonvalue_2d[ni,nj]` : longitude coordinates
  - `bounds_lat_2d[4,ni,nj]` : latitudes boundaries
  - `bounds_lon_2d[4,ni,nj]` : longitudes boundaries

- For unstructured domain
  - `(integer) nvertex` : max corners/edges of cells
  - `(double) latvalue_1d[ni]` : latitude coordinates
  - `(double) lonvalue_1d[ni]` : longitude coordinates
  - `(double) bounds_lat_2d[nvertex,ni]` : corners latitude coordinates
  - `(double) bounds_lon_2d[nvertex,ni]` : corners longitude coordinates
  - If cell nb corners < `nvertex`, corners must be cyclic redundant (CF compliance)

# The field element <field />

- Represent incoming or outgoing data flux from models

- Data can be sent or received at each time step from model throw 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/>
```

# Data fields 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 than declared in the grid
  - Example :

```
<grid id="grid_3d">
  <domain id="domain_2d" type="rectilinear" ni_glo="100" ni="10" data_ni="14"
                                            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 x data_nj x n_glo) = (14*7*20) = 1960`
  - Can be declared as :

  `REAL(kind=4) :: temp(14,7,20)`

Or `REAL(kind=4) :: temp(1960)` but data order must be `(14,7,20)` following the row major order Fortran convention

Or `REAL(kind=8) :: temp(1960)`

# FIELDS

## Field can be output to files

- Will appear as a child file element
- Construct files by including list of field required
- A field can appear, in more than one file
  - 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 related to file output

- 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** : fix **valid_min** & **valid_max** nectdf attribute

- Enable/disable field output
  - (boolean) **enabled** : if false, field will not be output (**default=true**)
  - (integer) **level** : fix the level of output of the field (**default=0**) with respect to "**level_output**" file attribute if (**level>level_output**) the field will not be output

- Setting missing values : fix masked point value in output file
  - (double) **default_value** : if not set, masked point value will be undefined

- Setting precision / 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**, usefull combine with **prec=2**
  - (integer) **compression_level (0-9)** : fix 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 not 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
  - **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" freq_output="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" freq_output="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)
  - (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 time CPU consuming since each value of the mesh must be tested independently

# Output files : the file element <file/>

## Defining fields to be written

- File elements can contains field elements or group field elements
- All listed field elements will be candidate to be 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 field_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

# FILES

## Enabling /disabling field 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
  - **(boolean)** **enabled** file attribute : if set to **false**, all fields are disabled
- Files with all fields disabled will not be output

## 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 sequential output (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 "**single_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 fix individually with field attribute

# VARIABLES

## Setting parameters : the variable element <variable/>

- Variable are used to define parameters
- Variable can be set or query from model
  - Could replace Fortran **namelist** or IPSL **run.def** files
- Used internally by XIOS (xios context) to define it own parameterization

### 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

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

# VARIABLES

## Setting or query value from model

- Set variable : `CALL 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>
</file>

-- from file --

USE xios
…
INTEGER :: int_var
CHARACTER(LEN=256) :: string_var
LOGICAL :: ierr

  ierr=xios_getvar('int_var',intvar)
  CALL 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"
```

# OUTPUT FILE STRUCTURE

## File structure

- XIOS respect 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 mixt instant and average time operation
  - Axis **time_instant** or **time_centred** may be written with the associated bounds

- Fields of different grids can be mixt 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

# OUTPUT FILE STRUCTURE

- 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 xios" ;
                :title = "Created by xios" ;
                :Conventions = "CF-1.5" ;
                :production = "An IPSL model" ;
                :timeStamp = "2015-Dec-14 15:20:26 CET" ;
```

# OUTPUT FILE STRUCTURE

LSCE

## Adding specific metadata

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

# OUTPUT FILE STRUCTURE

## Appending data to an existing file

- When restart models, field data can be appended to a previous xios output files
- **(boolean)** **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

- In order to avoid biggest file, file can be split periodically
- File will suffixed with start date and end date period
- **(duration)** **split_freq** : split file at **split_freq** period
- Can be combine with append mode

## 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

## Reading data from file

- Very new XIOS 2 functionalities
  - Specifications and support will be improved in closed future
- (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
- 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

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

-- From model ---

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

- Field with no time record will be read only once

# PARALLEL WORKFLOW FUNCTIONNALITIES

## Why Workflow ?

- Field are exposed from model at each time step
  - internally represent 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 filter can be chained together to achieve complex treatment
- All filters are all parallel and scalable
- 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

## Workflow end point

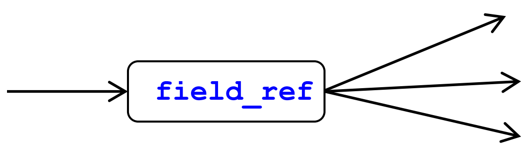
- Output flux can be sent to servers and written to file
- Output flux can be read from model (**xios_recv_field**)
  - **(boolean)** **read_access** field attribute : field read from models must set **read_access**="**true**"
  - Field directly read from file have automatically **read_access**="**true**"

# PARALLEL WORKFLOW FUNCTIONNALITIES

```
<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>

-- From model ---

DO ts=1,n
  CALL xios_update_timestep(ts)
  CALL xios_send_field("precip",precip)
  CALL xios_recv_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 kind of filters

- Arithmetic filters : combine flux together
- Time integration filters : integrate flux over a period
- Spatial filters : modify 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>
```

# FILTERS

- 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`", "`cumulate`"
- 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 the time integration is done
  - Generated flux is the sent to servers to be written

LSCE

# FILTERS

- Flux will is 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

```
<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" freq_output="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>
        --from model--
CALL xios_send_field("temp",temp)
```

## Chaining and combine time filters and arithmetic's filters

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

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

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

# Spatial filters

- Spatial filters may change the geometry, dimensionality and the parallelism data distribution of a flux
- Algorithms must parallel and scalable in order to perform the flux transformation on whole allocated parallel resources of a simulation
- Difficult to develop, such features are still experimental in XIOS 2
- Actually, limited number of filters, more such parallel filters will be develop in future

## Using spatial filter

- Spatial filters are enabled when the grid of a referenced field is different of the current grid field
  - No spatial filter enabled (same grid ref)

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

  - Try to enabled spatial filter
  - (different grid ref)

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

- 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

# FILTERS

- To find which filters 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

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

<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_unstructured">
  <domain domain_ref="unstructured">
    <interpolate_domain/>
  <domain/>
  <axis axis_ref="vert_axis" >
</grid>
```

- Implement interpolation filter between "**regular**" domain and "**unstruct**" domain, no filter implemented between axis

# FILTERS

- 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"

# Spatial filters actually developed

## Zooming filters

- Extract sub-part of data
- Zoom on axis
  - **zoom_axis** transform
  - **(integer) begin** attribute : global beginning index of the zoom
  - **(integer) n** attribute : global size of the zoom
- Zoom on domains
  - **zoom_domain** transform
  - **(integer) ibegin** attribute : global beginning index of 1st dimension of the zoom
  - **(integer) ni attribute** : global beginning index of 1st dimension of the zoom
  - **(integer) jbegin** attribute : global beginning index of 2nd dimension of the zoom (for rectilinear or curvilinear)
  - **(integer) nj** attribute : global beginning index of 2nd dimension of the zoom (for rectilinear or curvilinear)

# FILTERS

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

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

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

<grid id="grid_zoom">
   <domain domain_ref="regular">
      <zoom_domain ibegin="20" ni="50" jbegin="100" nj="60" />
   <domain/>
   <axis axis_ref="vert_src">
      <zoom_axis begin="30" n="40"/>
   </axis>
</grid>
```

- Zoom of global size `(50,60,40)` starting at indices `(20,100,30)`
- Only this part will be output to files

# FILTERS

## Inverse axis data and coordinates

- `inverse_axis` transformation

## Interpolate/remap axis and domain

- Axis interpolation
  - Actually only polynomial interpolation
- `interpolate_axis` transformation
  - `(integer) order` attribute : set the order of the polynomial interpolation
  - Ex:

```
<axis n_glo="100">
    <interpolate_axis  order="2"/>
</axis>
```

- Domain interpolation
  - 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 $1^{st}$ or $2^{nd}$ order
- `interpolate_domain` transformation
  - `(integer) order` attribute : set the order of the conservative interpolation (1 or 2)
  - Ex :

```
<domain ni_glo="10000" type="unstructured">
    <interpolate_domain  order="2"/>
</domain>
```

# FILTERS

## Generating missing attribute on regular rectilinear grid

- Still experimental
- Generate automatically parallel distribution
- Generate automatically longitude and latitude values

- **generate_rectilinear_domain** transform
  - (double) lon_start, lon_end, lat_start, lat_end attr: in north degrees, nothing means global grid
  - (double) bounds_lon_start, bounds_lon_end, bounds_lat_start, bounds_lat_end attr : for boundaries version
  - Useful to perform automatic interpolation on regular grid

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

<grid id="grid_unstructured">
  <domain domain_ref="unstruct" />
</grid>

<grid id="grid_regular">
  <domain domain_ref="regular/>
     <generate_rectilinear_domain/>  <!-- Create automatically 1° resolution regular domain -->
      <interpolate_domain/>          <!-- and remap -->
  <domain/>
</grid>

<field id="temp_unst"                          grid_ref="unstruct"/>
<field id="temp_reg" field_ref="temp_unst"  grid_ref="grid_regular"/>
```

## 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_zoom" field_ref="temp_reg"         grid_ref="grid_regular_zoom"/>
```

- To avoid intermediate field definition, use `grid_path` attribute
  - `(string) grid_path` attribute : define the list of intermediate grid

```
<field id="temp_unstr"                                                        grid_ref="grid_unstruct"/>
<field id="temp_reg_zoom" field_ref="temp_unstr" grid_path="grid_regular" grid_ref="grid_regular_zoom"/>
```

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

```
<field id="temp_unstr" domain_ref="unstructured" />
<field id="temp_reg_zoom" field_ref="temp_unstr" domain_ref="regular_zoom"/>

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

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

# FORTRAN INTERFACE

```
SUBROUTINE client(rank,size)
  USE xios
  IMPLICIT NONE
  INTEGER :: rank, size
  TYPE(xios_time)      :: dtime
  DOUBLE PRECISION,ALLOCATABLE :: lon(:,:),lat(:,:),a_field(:,:)
  ! other variable declaration and initialisation .....

! XIOS initialization
  CALL xios_initialize("client", return_comm=comm)
  CALL xios_context_initialize("hello_word",comm)
! Complete horizontal domain definition
  CALL xios_set_domain_attr("horizontal_domain",ibegin=ibegin,ni=ni,jbegin=jbegin, nj=nj)
  CALL xios_set_domain_attr("horizontal_domain ",lonvalue_2d=lon, latvalue_2d=lat)
! Setting time step
  dtime%second=3600
  CALL xios_set_timestep(dtime)
! Closing definition
  CALL xios_close_context_definition()
! Entering time loop
  DO ts=1,96
    CALL xios_update_calendar(ts)
    CALL xios_send_field("a_field",a_field)
  ENDDO
! XIOS finalization
  CALL xios_context_finalize()
  CALL xios_finalize()
END SUBROUTINE client
```

# Fortran structure to be XIOS compliant

- For an exhaustive description of XIOS Fortran API : see xios reference guide

## XIOS Initialization

- 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

- `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
- Several different context can be initialized during same run
- All xios fortran call are collective for the related current context MPI communicator

## Switching to a context

- `CALL set_current_context("context_id")`
  - All behind xios fortran call will be related to context `"context_id"`

# FORTRAN INTERFACE

## Complete the XML database definition

- Fix missing attribute
  - Some attribute values are known only at run time
- All attribute can be fixed use 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

## Closing context definition

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

# FORTRAN INTERFACE

## Entering time loop

- 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
  - Sent field request are not taking into account 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 that the related time step

## Finalize context

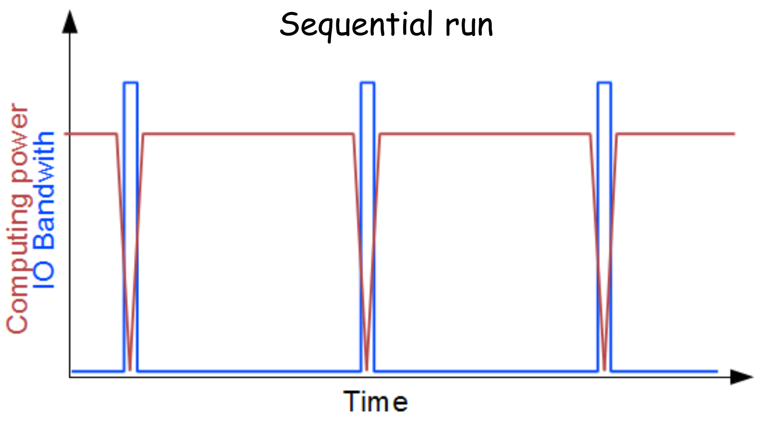
- All opened context must be finalized after the end of time loop
- CALL xios_context_finalize

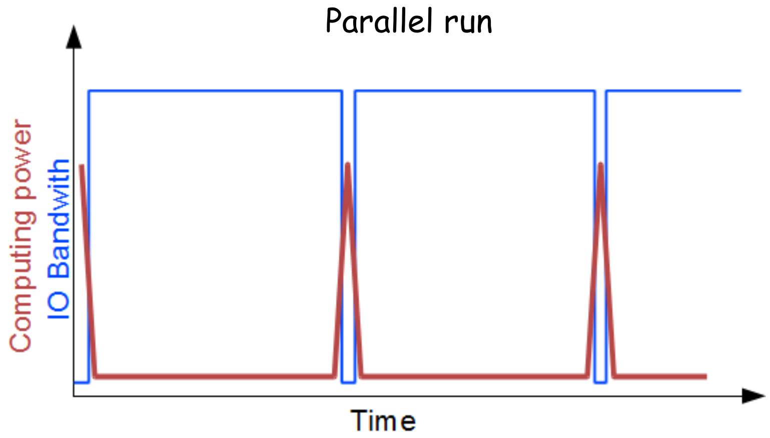
## Finalize xios

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

# THE I/O Problem

## 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 of the file system, performance may breaking down when attempt 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

# THE I/O PROBLEM

LSCE

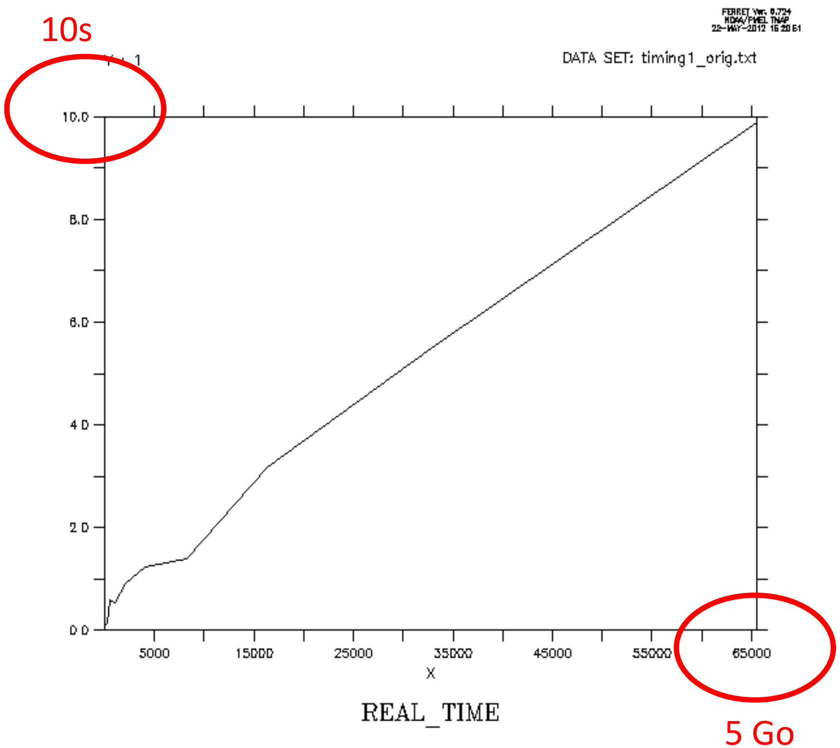
- Using parallel IO ?
  - Best way to achieve scalable IO without rebuild file
  - But difficult to aggregate a lot of I/O bandwidth with a big number of writing processes
  - Parallel IO are very less scalable than models due to hardware restriction (pricy and not took into account for performance evaluation)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

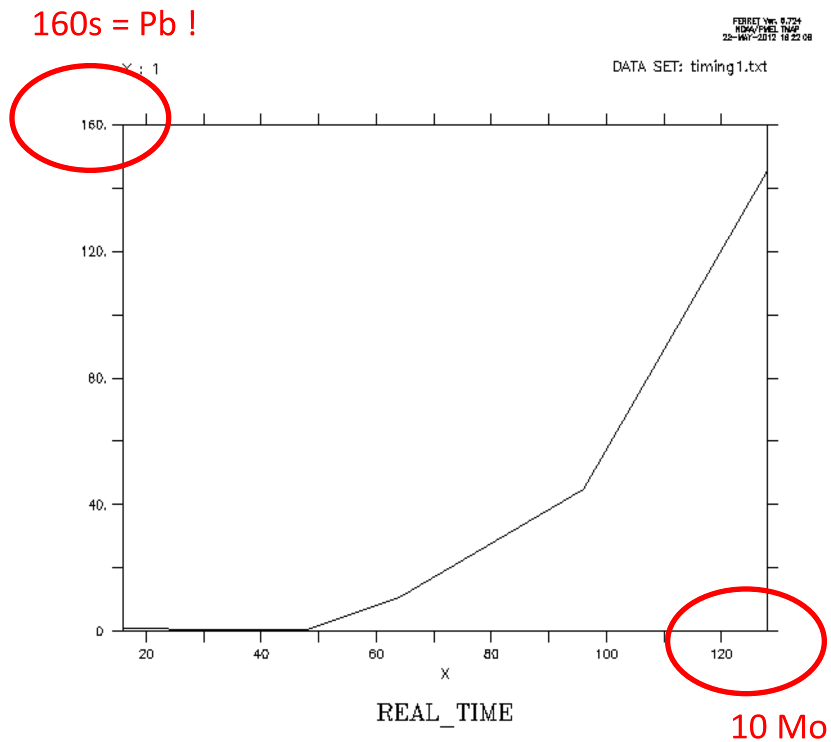
- Using asynchronous parallel IO ?
  - Good way to overlap IO by computing
  - MPI/IO : difficult to manage, load balancing problem…
  - High level library (HDF5, netcdf…) generally don't implement asynchronous IO.

- 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

# THE I/O PROBLEM

- 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

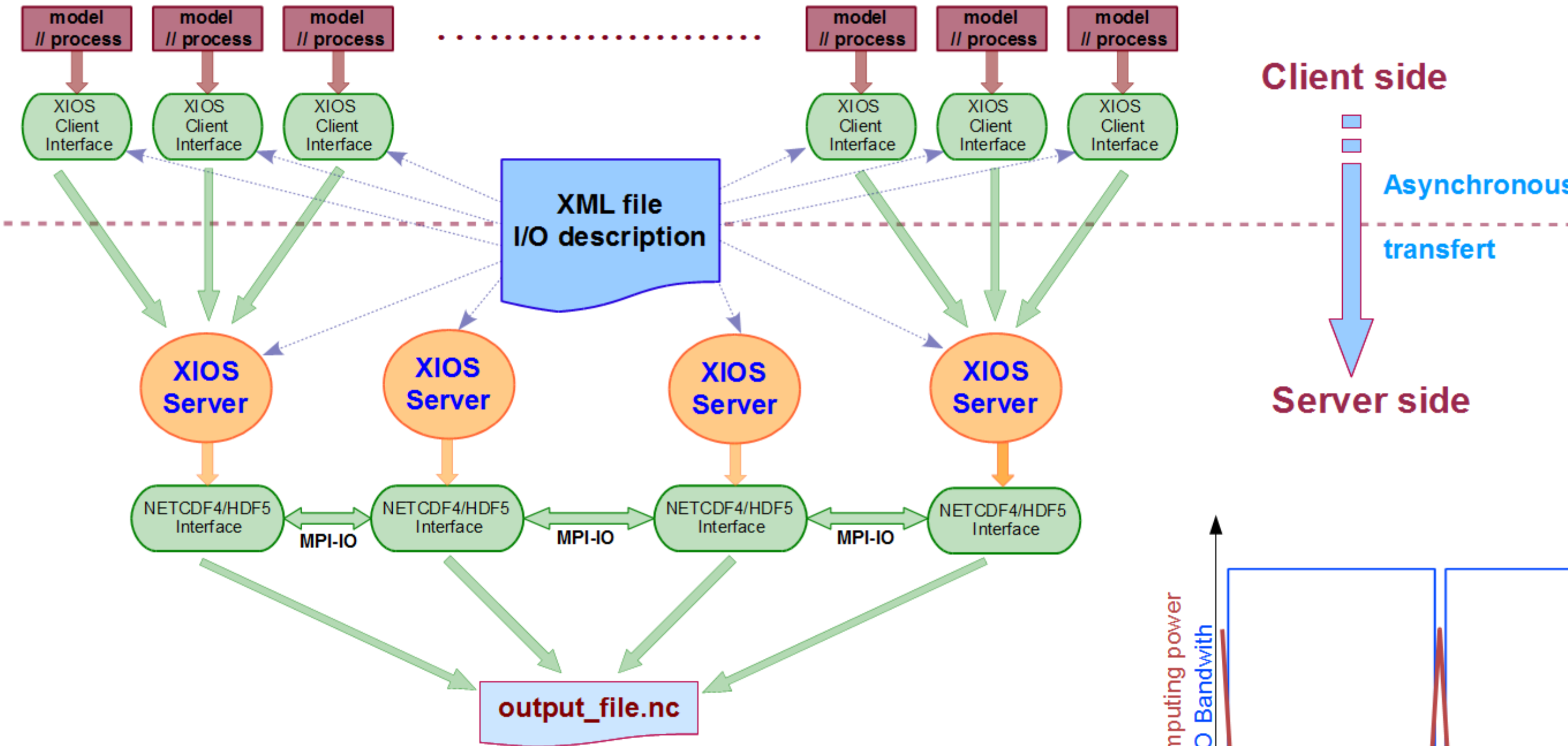


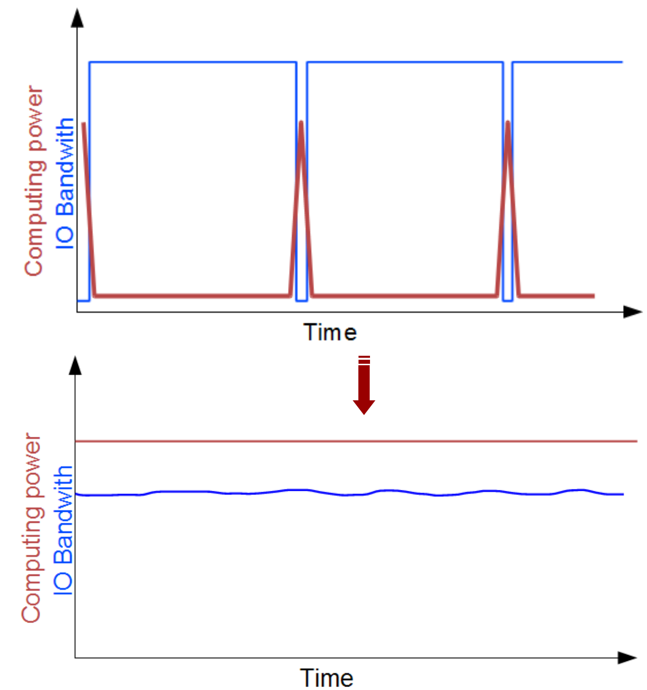




–*Multiple file on 16 CPUs : 1 file by process = 16 files*

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

# XML I/O SERVER FUNCTIONNALITIES



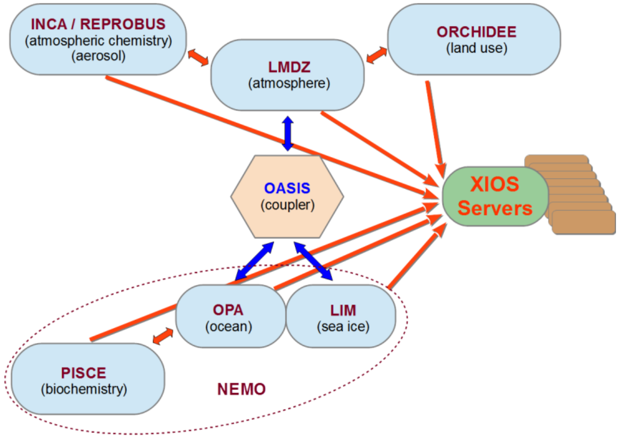

## XIOS servers

## 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

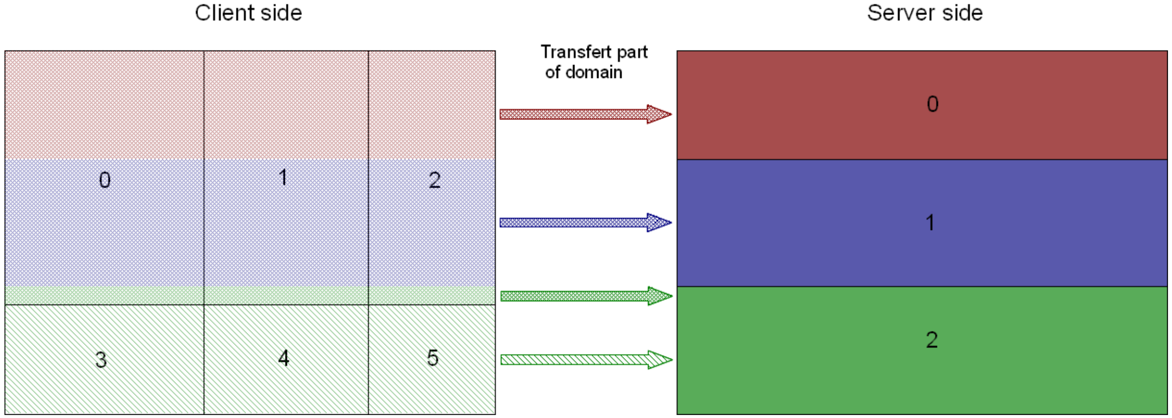
# Complex and fully asynchronous protocol

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




## Same pools of I/O servers used in coupled model

## Different data distribution between client and servers



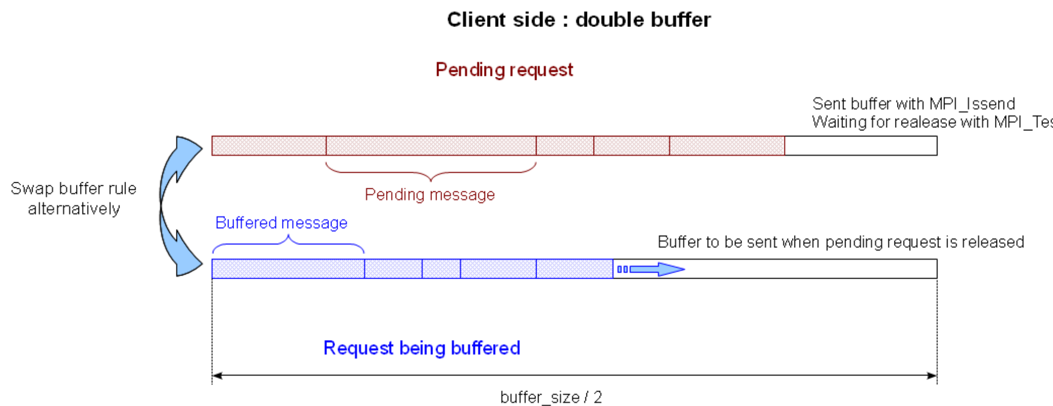

## Data are sent asynchronously at writing time

- 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

## Data are received asynchronously with prefetching (by advance) on client side

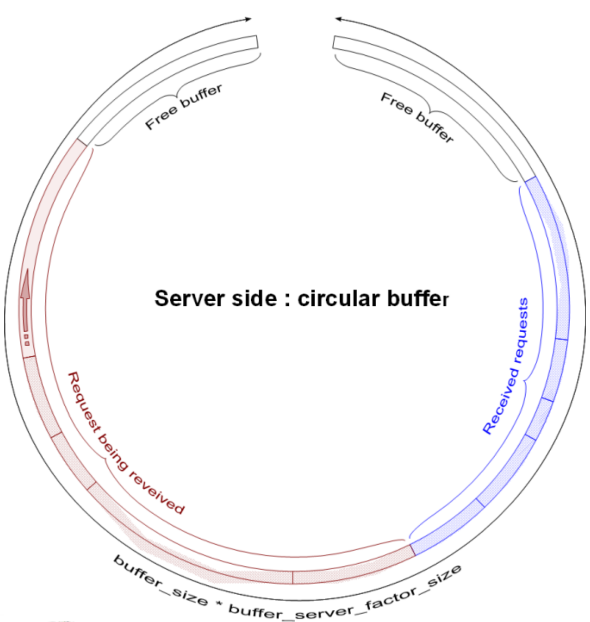
## Large usage of buffers

- 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**

# UNDERSTANDING I/O SERVERS USAGE

## Understand and analyze XIOS servers performance

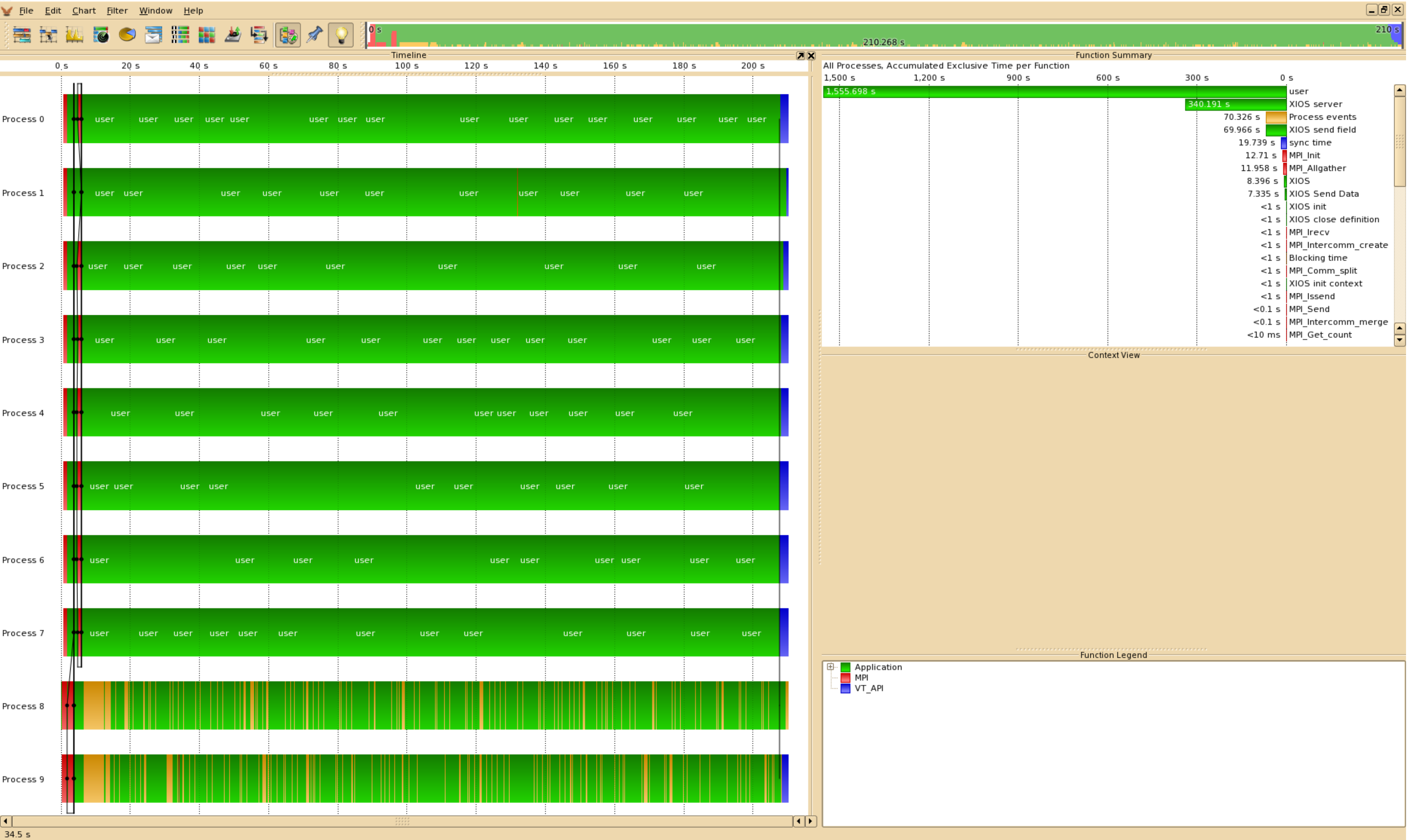
- Build a toy model
- Field is sent and written at each time step
- Some extra working time is simulate by a waiting call

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

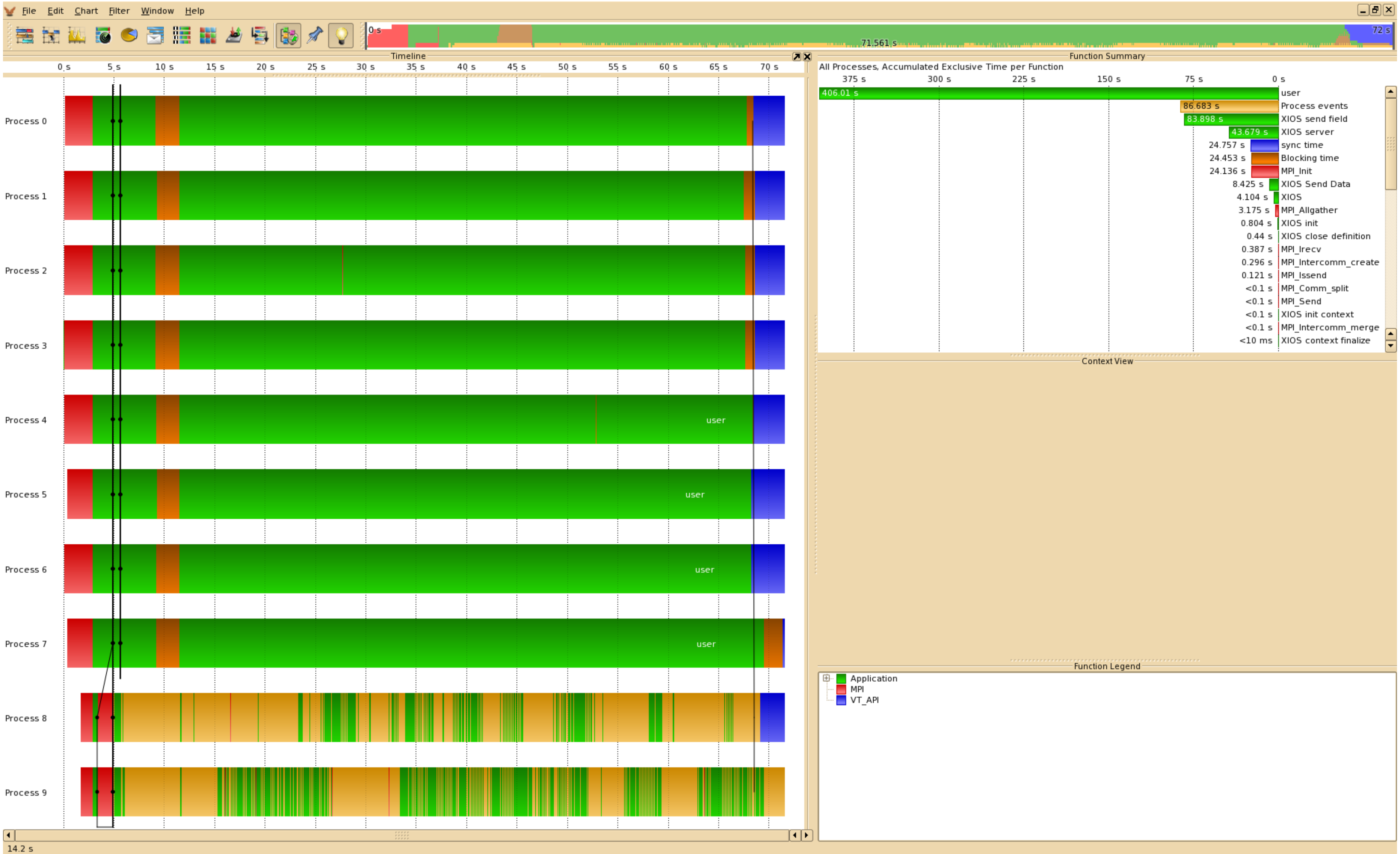
- Look at parallel vampir trace
  - Green : application time
  - Red : MPI function time
  - Orange : server working time
  - Brown : client waiting for free buffer and blocking

- Make experiments by decreasing working time compared to I/O output

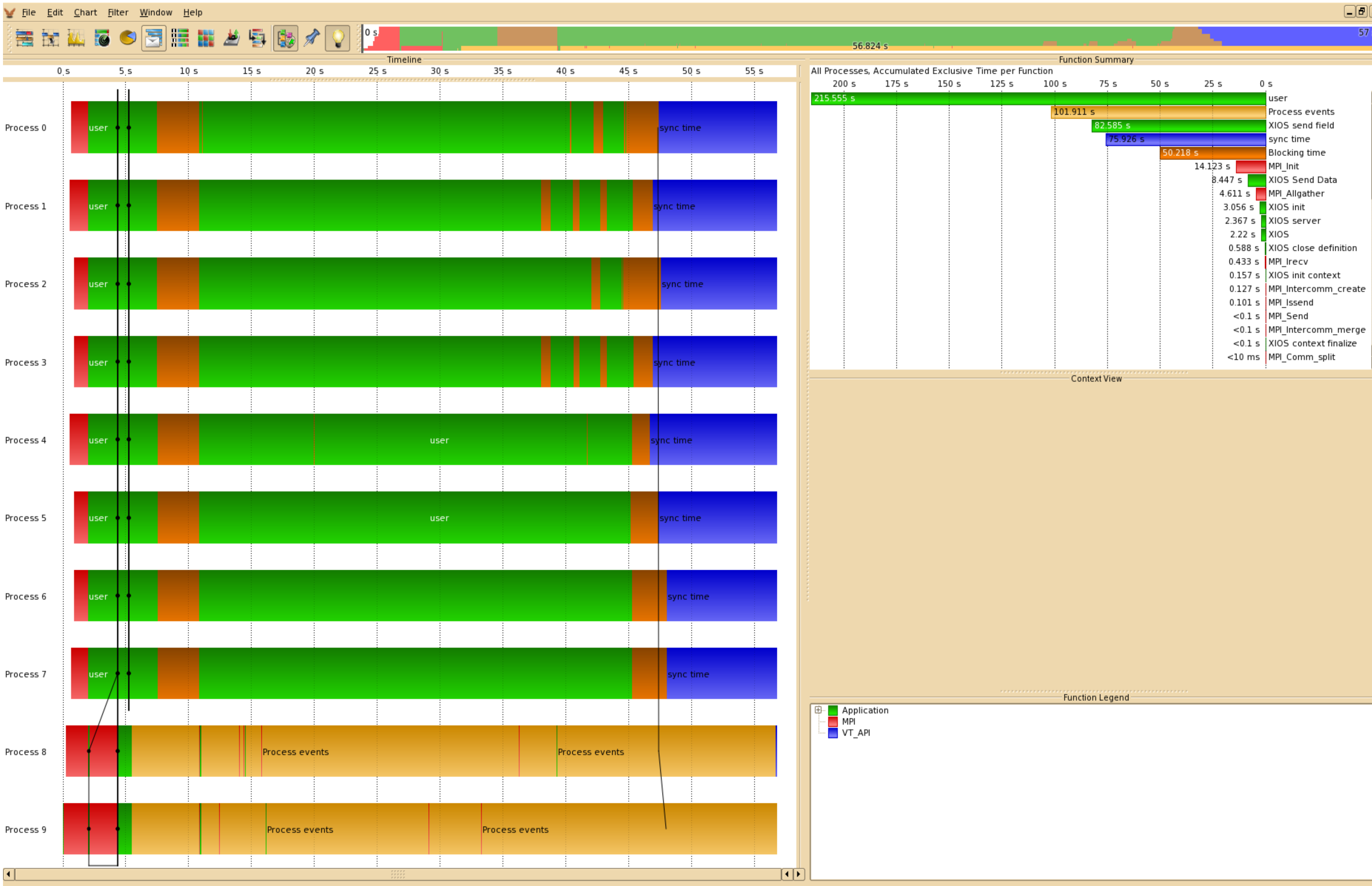
# 8 clients – 2 servers : working time by iteration: 80 ms



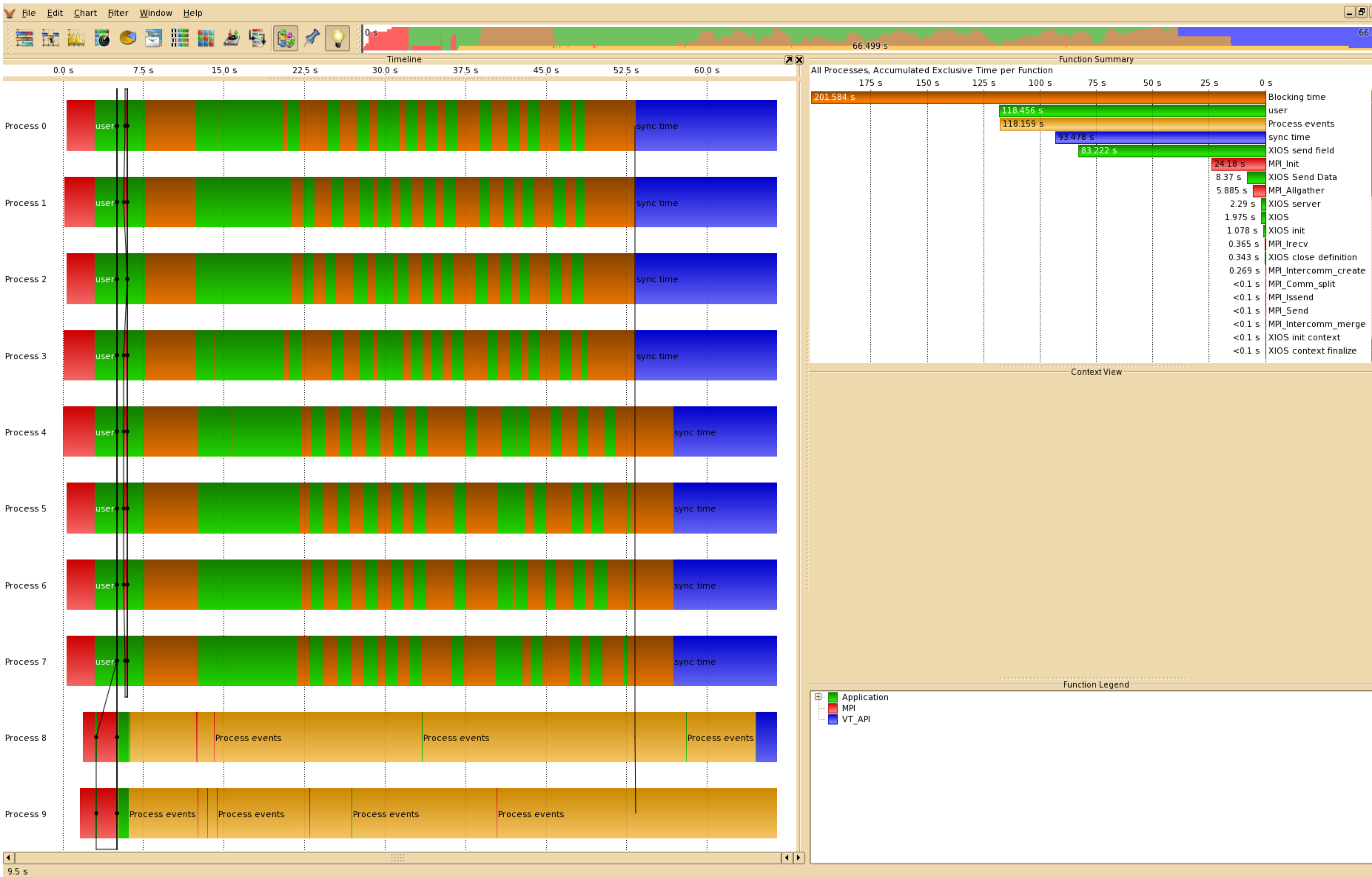
# 8 clients – 2 servers : working time by iteration: 20 ms



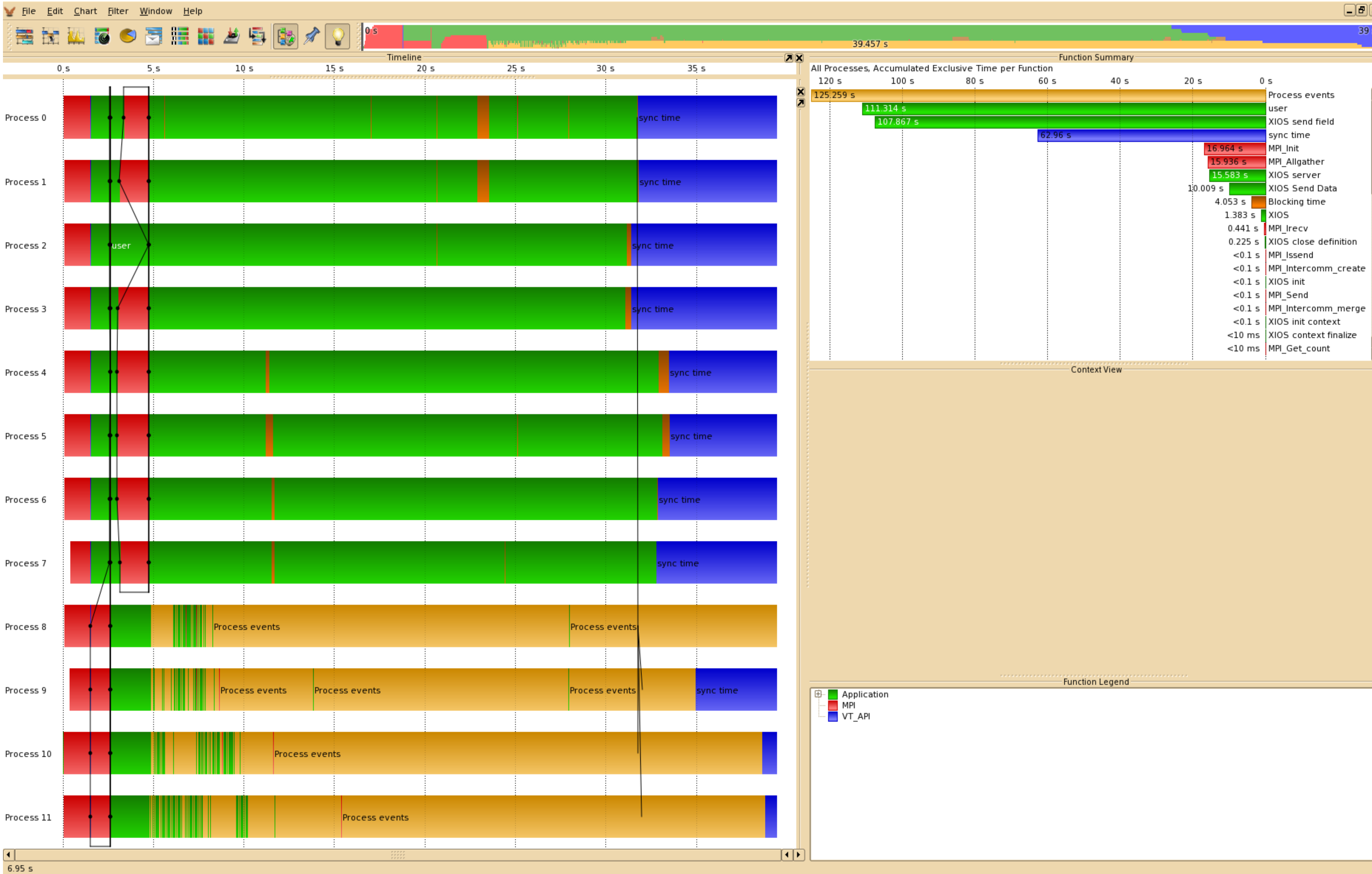
# 8 clients – 2 servers : working time by iteration: 10 ms



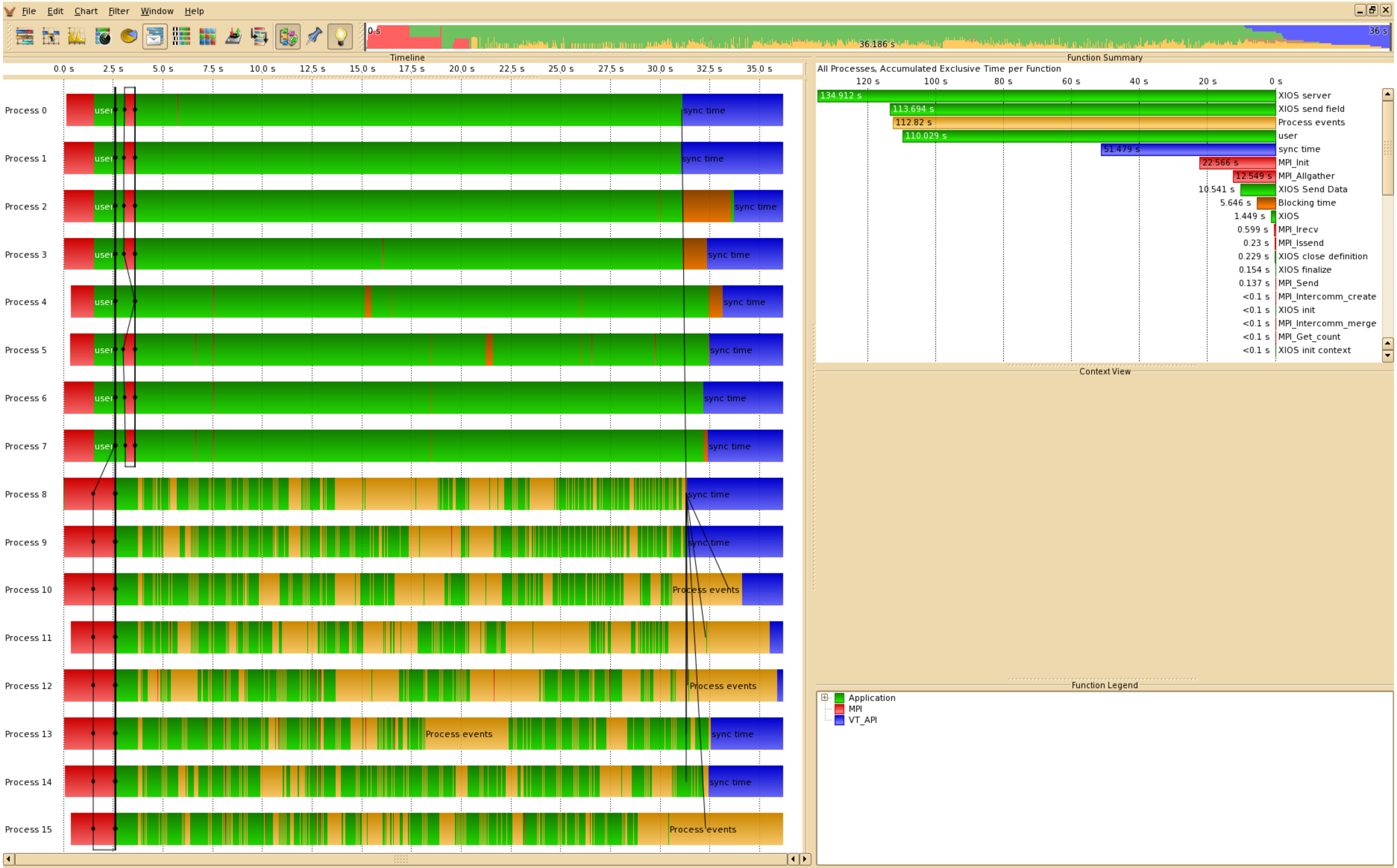
# 8 clients – 2 servers : working time by iteration: 5 ms



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



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



# 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

# PERFORMANCE

## 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
  - XIOS act only as a library
- Each client is itself a server for the other clients
  - Pool of servers is equal to the number of clients
- Synchronous mode only
  - Client must wait that sent data is written before continue
- "Single file" mode : all client process attempt to write parts of file
  - Bad for performance for high number of clients
- "Multiple file" mode : one file by client
  - Difficult to rebuild in post-treatment

## Memory consumption

- XIOS consume memory internally to make averaging and other operations
- XIOS use large transfer buffer for asynchronous protocol
- Part of memory is all consumed by NETCDF4/HDF5
- But generally, memory consumption is scalable
  - Increasing number of clients decrease memory consumption on client side
  - Increasing number of servers decrease memory consumption on server side

## Memory report

- Give informations about memory used by transfer buffer
- Buffer size is automatically computed
  - Can be different for each communication channel client-server couple
  - Dependent of the parallel data distribution
- For each couple client-server, 2 channels, 2 different buffers
  - 1 for sending/receiving data client -> server (I/O write)
  - 1 for sending/receiving data server -> client (I/O read)

# PERFORMANCE

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 loosing 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` parameter

## 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 x 2882 x 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 parameterization

- 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"

# XIOS PARAMETRIZATION

- (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
  - Usefull 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`)