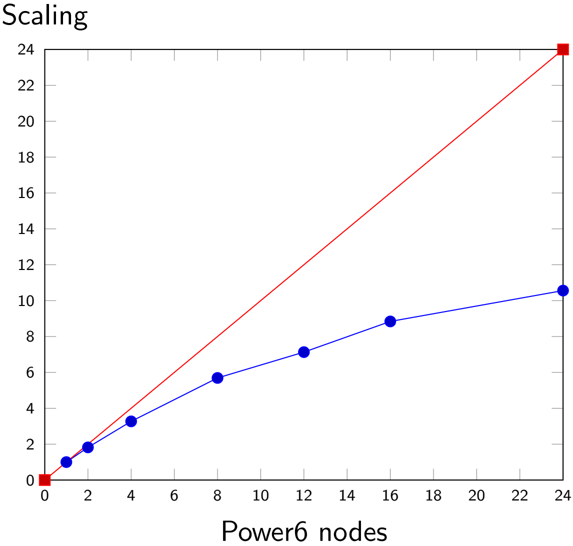
Parallel I/O in climate and NWP

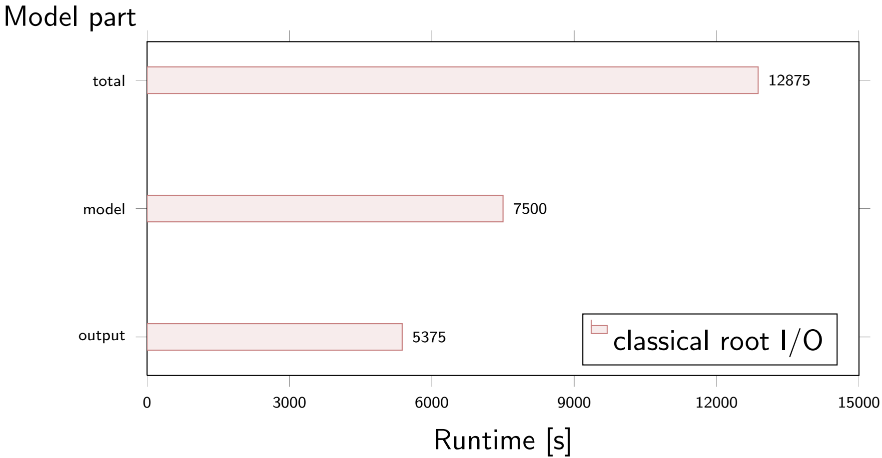
Luis Kornblueh

with contributions from
Deike Kleberg and Uwe Schulzweida, MPI for Meteorology
Mathias Pütz, CRAY
Christoph Pospiech, IBM
Thomas Jahns, Moritz Hanke, Mathis Rosenhauer and Jörg Behrens, DKRZ
and
Yann Meurdesoif, Arnaud Caubel, Sebastian Masson, IPSL

Scaling optimization result for echam6, T63L47



Classical root I/O explains scaling



Atmospheric and Oceanographic Data

Requirements

- long term metadata and data storage
- standardization
- compression

Solutions

- WMO GRIB standard
- lowest entropy data subsampling
- two stage compression: lossy entropy based and lossless compression of resulted *image* (CCSDS recommendation based, DKRZ implemented libaec, replaces szip) — metadata uncompressed!

I/O improvements possible

Improvement by additional packing of the BDS data

Resolution	GRIBZip2d	grib-aec	gzip (external)
Source	Frauenhofer	MPI-M	GNU
T42 L19	2.32	2.13	2.06
T63 L31	1.85	1.78	1.35
T106 L60	5.17	4.75	3.81
T213 L31	3.09	3.06	2.41
mean	3.03	2.93	2.15

I/O improvements possible

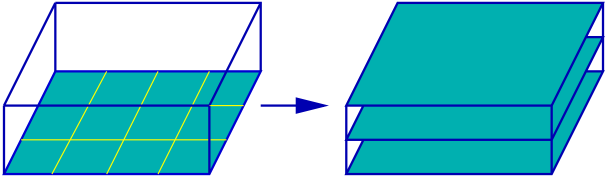
Improvement by additional packing of the BDS data

Resolution	GRIBZip2d	grib-aec	gzip (external)
Source	Frauenhofer	MPI-M	GNU
T42 L19	2.32	2.13	2.06
T63 L31	1.85	1.78	1.35
T106 L60	5.17	4.75	3.81
T213 L31	3.09	3.06	2.41
mean	3.03	2.93	2.15

Remark

netCDF stores 4 byte, grib in average 2 byte — compression ratio given with respect to the later.

A transposition problem



Problem

- model decomposition is based on one- or two-dimensional horizontal slicing
- storage unit of model data is based on vertical slicing
- requires transpose and data gathering

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose
5. compress each individual record

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose
5. compress each individual record
6. write ... sort of

Solution strategy

1. decompose I/O in a way that all variables are distributed over the collector/concentrator PEs (I/O PEs)
2. store each compute PEs data in buffers to be collected by collector PEs (I/O PEs) — buffer should reside in RDMA capable memory areas
3. instead of doing I/O, copy data to buffer and continue simulation
4. collectors collect their respectable data (gather) via one-sided (RDMA based) MPI calls and do the transpose
5. compress each individual record
6. write ... sort of

First five steps runs fine, but the last one makes a lot of trouble!

File writing in ECHAM TO-BE

- After calculating one I/O timestep the compute processes copy their data to a buffer and go on calculating till the next I/O timestep.
- I/O processes fetch the data using MPI one sided communication.
- Gather and transpose of the data is based on callback routines supplied by the model.

Most important property

Compute processes are not disturbed by file writing.

Known difficulties

- single offload step requires large memory on offload-node (requires eventually changes for Linux cluster and Cray XT architecture type machines, and maybe for IBM BlueGene)
- generates network jitter (RMA access to all compute nodes concurrent with computing nodes internal communication)
- filesystem jitter due to system bottlenecks (eg. blizzard@dkrz: total bandwidth 30 GB/s, 2 GB/s per node, but 256 nodes)

What to optimize?

Search strategy

1. get to know your systems I/O capabilities!

What to optimize?

Search strategy

1. get to know your systems I/O capabilities!
2. measure the I/O bandwidth achieved

What to optimize?

Search strategy

1. get to know your systems I/O capabilities!
2. measure the I/O bandwidth achieved
3. build a test case for your I/O library

What to optimize?

Search strategy

1. get to know your systems I/O capabilities!
2. measure the I/O bandwidth achieved
3. build a test case for your I/O library
4. *profile* your testcase

What to optimize?

Search strategy

1. get to know your systems I/O capabilities!
2. measure the I/O bandwidth achieved
3. build a test case for your I/O library
4. *profile* your testcase
5. track down to *offending* level

What to optimize?

Search strategy

1. get to know your systems I/O capabilities!
2. measure the I/O bandwidth achieved
3. build a test case for your I/O library
4. *profile* your testcase
5. track down to *offending* level
6. check selected counters for *offending* code part

I/O capabilities

An example: DKRZ

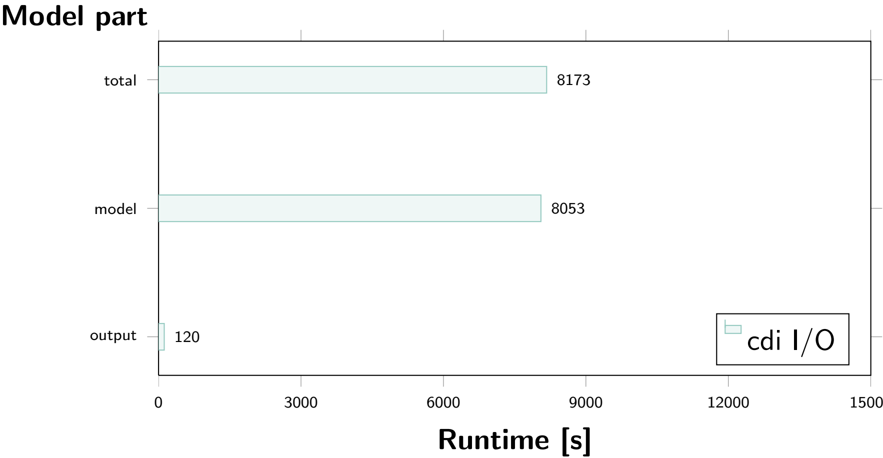
- 256 compute nodes, 12 file server, 6 PB filesystem, 4 HPSS server, 60 PB tape archive
- total I/O bandwidth to disk 30 GB/s
- per node max. I/O bandwidth 2 GB/s
- 1600 users
- max. 96 concurrent post-processing jobs
- unknown number of production jobs

Analysis for optimization strategy

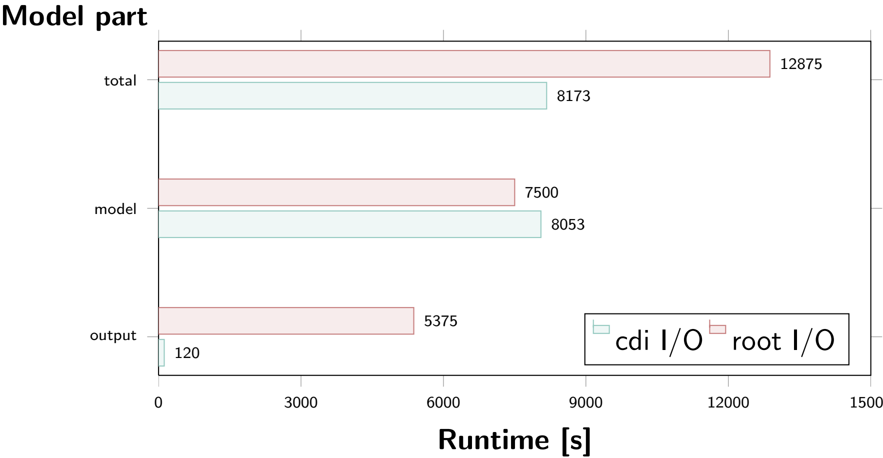
Caution: assembler reading required

- understand roughly how your CPU works
- need to *read* assembler (not that bad, feels like using a pocket calculator), you get an idea what the compiler is doing
- compare code of different optimization levels
- try to find the patterns, you would expect for fast code

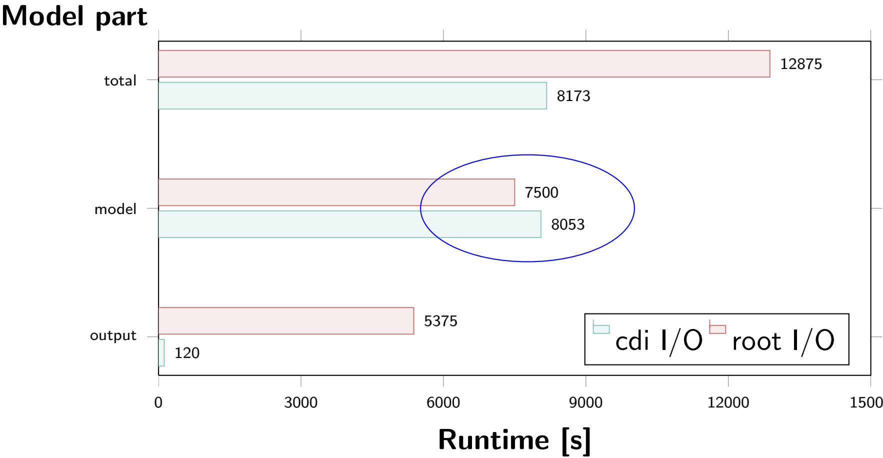
Compare


Model part
total
model
output
8173
8053
120
cdi I/O
0
3000
6000
9000
12000
1500
Runtime [s]

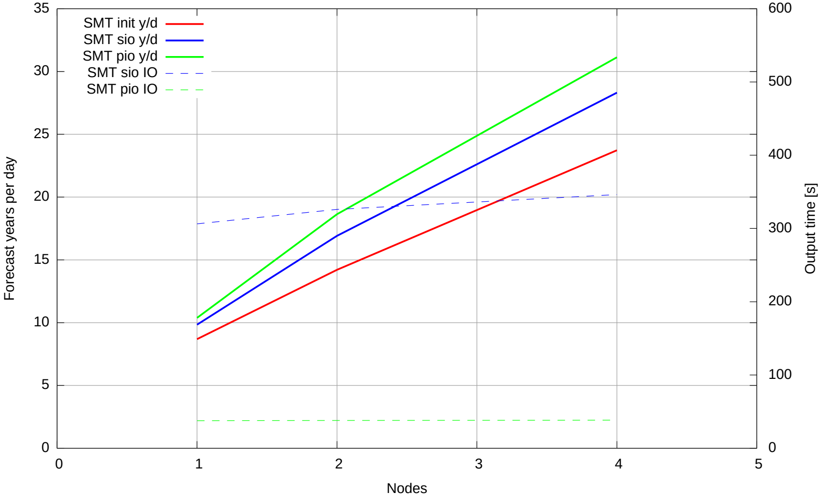
Compare


Model part
total
model
output
12875
8173
7500
8053
5375
120
cdi I/O
root I/O
0
3000
6000
9000
12000
1500
Runtime [s]

Compare


Model part
total
model
output
12875
8173
7500
8053
5375
120
cdi I/O
root I/O
0
3000
6000
9000
12000
1500
Runtime [s]

Real application improvements



Some recommendations on data handling

- do not duplicate data

Some recommendations on data handling

- do not duplicate data
- compress domain specfic, data only to allow fast metadata access

Some recommendations on data handling

- do not duplicate data
- compress domain specfic, data only to allow fast metadata access
- keep large files, maybe use containers

Some recommendations on data handling

- do not duplicate data
- compress domain specfic, data only to allow fast metadata access
- keep large files, maybe use containers
- use the precision customized for your problem (information entropy tells you how much bits needed)

Some recommendations on data handling

- do not duplicate data
- compress domain specfic, data only to allow fast metadata access
- keep large files, maybe use containers
- use the precision customized for your problem (information entropy tells you how much bits needed)
- never, ever move data per se (copy), always combine with data reduction or other transformations needed

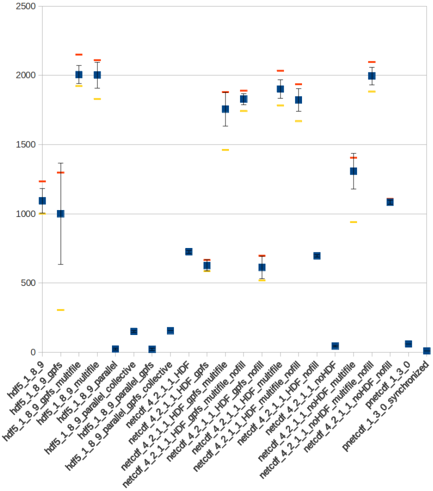
Some recommendations on data handling

- do not duplicate data
- compress domain specfic, data only to allow fast metadata access
- keep large files, maybe use containers
- use the precision customized for your problem (information entropy tells you how much bits needed)
- never, ever move data per se (copy), always combine with data reduction or other transformations needed
- move second level data away from the HPC production machine

Some recommendations on data handling

- do not duplicate data
- compress domain specfic, data only to allow fast metadata access
- keep large files, maybe use containers
- use the precision customized for your problem (information entropy tells you how much bits needed)
- never, ever move data per se (copy), always combine with data reduction or other transformations needed
- move second level data away from the HPC production machine
- develop a new concept replacing file systems by something more science problem aware (*data blocks, neighborhood relations, meta data associated with, raw disk block usage)*

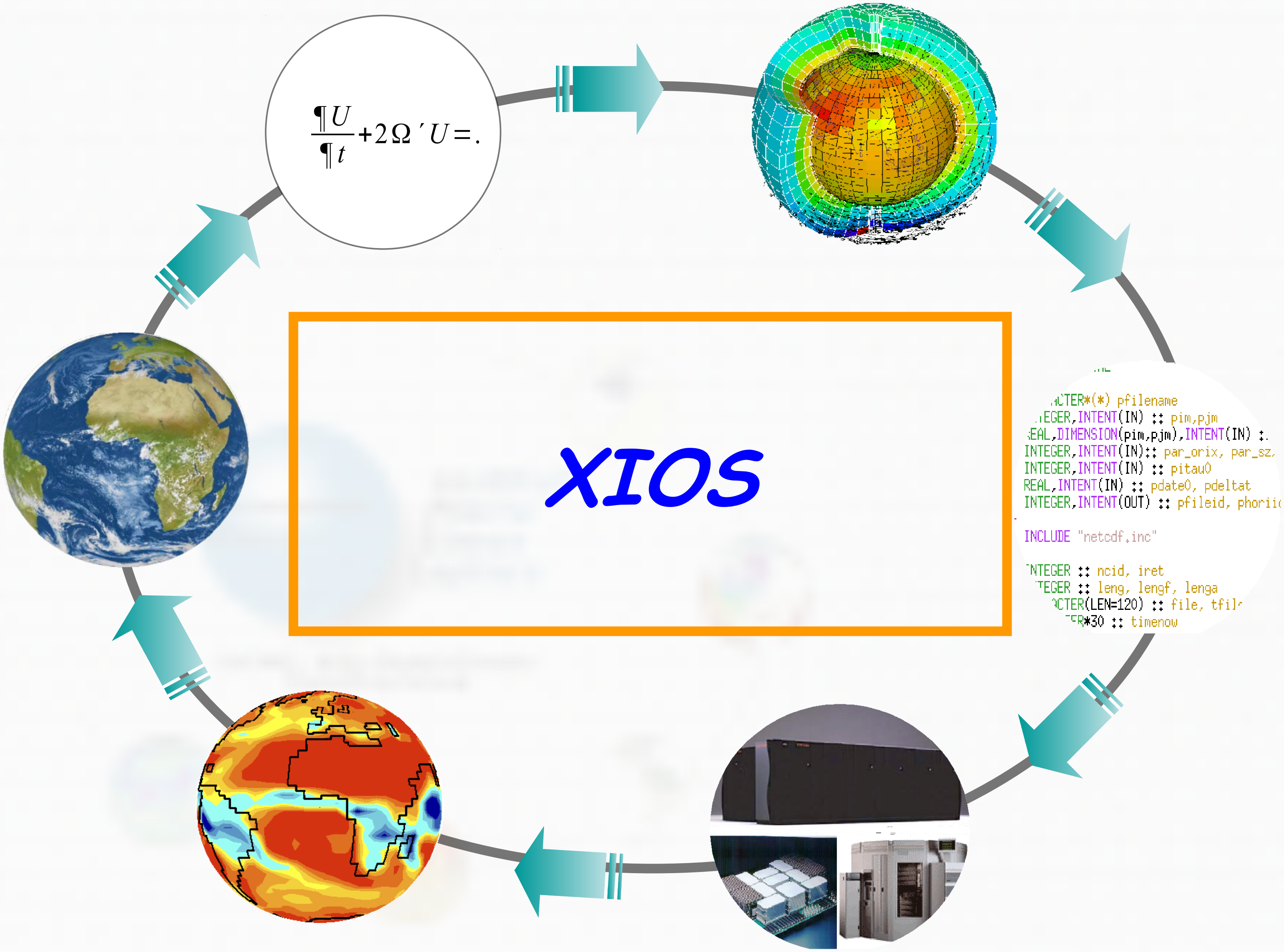
What to do with this?



Courtesy: Nathanael Hübbe, University of Hamburg, DKRZ

Yann Meurdesoif(LSCE-IPSL), A. Caubel, S. Masson

XIOS



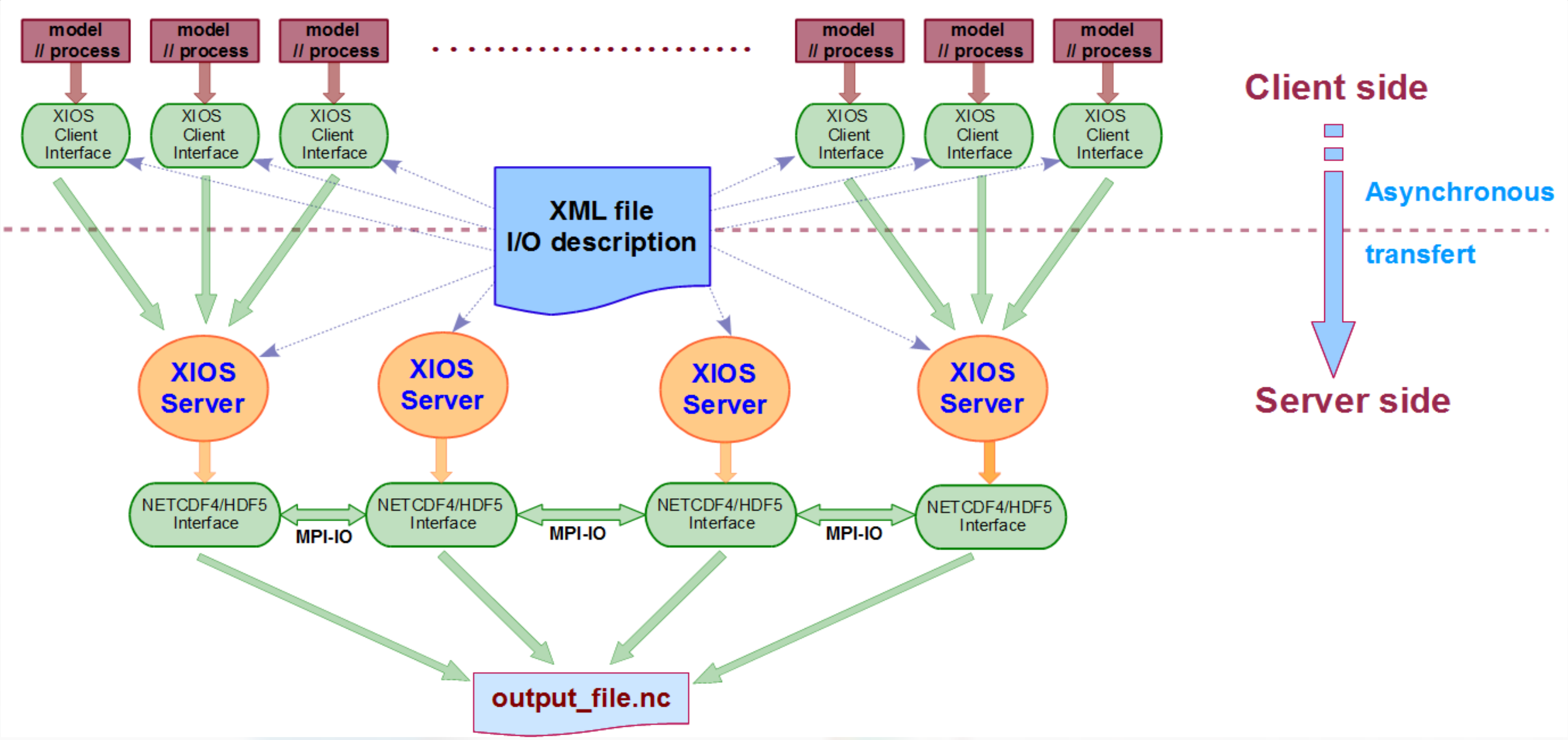
$$\frac{\P U}{\P t}+2\Omega\,'\,U=.$$

```
  ACTER*(*) 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"

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

XIOS - Motivation



- XIOS stands for XML – IO – SERVER
  - Library developed at IPSL and dedicated to IO management of climate model.
    - Management of output diagnostic, history file.
    - Temporal post-processing operation (averaging, max/min, instant, etc…)
    - Spatial post-processing operation.
  - Rewrote in C++ and improved as a part of the IS-ENES project.
    - ~ 35 000 code lines under SVN : http://forge.ipsl.jussieu.fr/ioserver/browser/XIOS/trunk

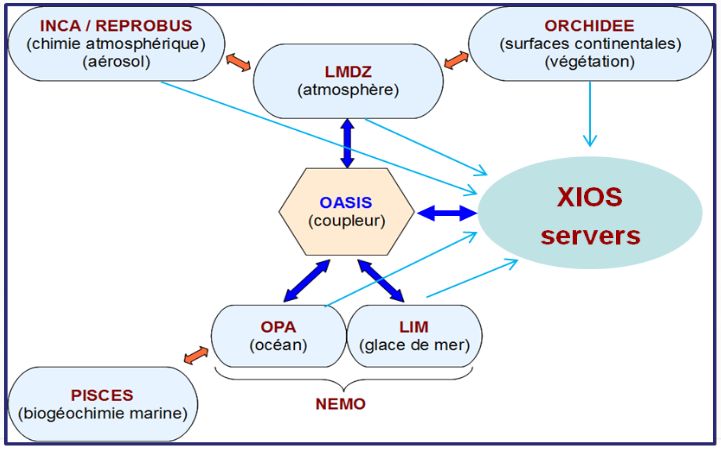
XIOS main goals

Flexibility

- Simple IO management in the code
  - Minimize calling subroutines related to IO definition (file creation, axis and dimensions management, adding and output field…)
  - Minimize arguments of IO calls.
- Ideally : output a field requires only an identifier and the data.
  - `CALL send_field(“field_id”, field)`
- Outsourcing the output definition in a XML file
  - Hierarchical management of definition with inheritance concept
  - Simple and more compact definition, avoid unnecessary repetition
- Changing IO definitions without recompiling
  - Everything is dynamic, XML file is parsed at runtime.

Performance

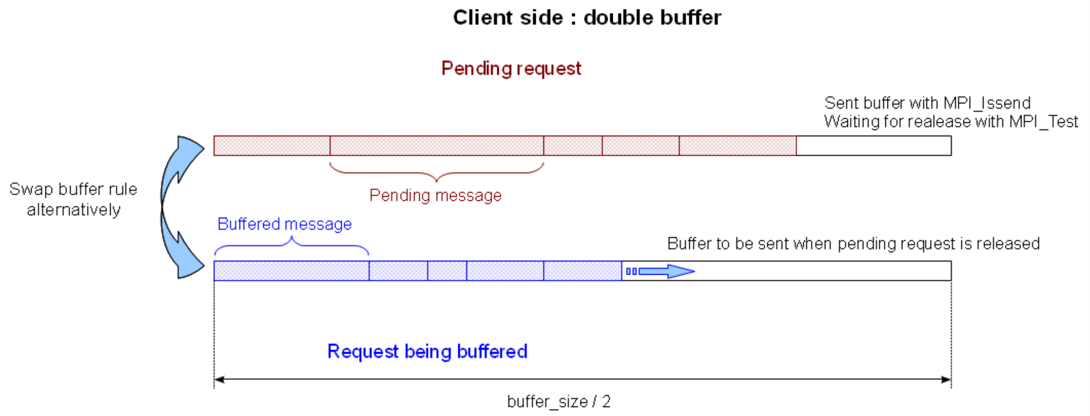
- Targeted for large core simulation (> ~10 000) on climate coupled model.
- Writing data must not slow down the computation.
  - Simultaneous writing and computing based on asynchronous call.
- Using one or more “server” processes dedicated exclusively to the IO management.
  - Asynchronous transfer of data from clients to servers.
  - Asynchronous data writing by each server.
- Use of parallel file system (Netcdf4-HDF5 format).
  - Simultaneous writing in a same single file by all servers
  - No more post-processing rebuild of the files



Communication protocol

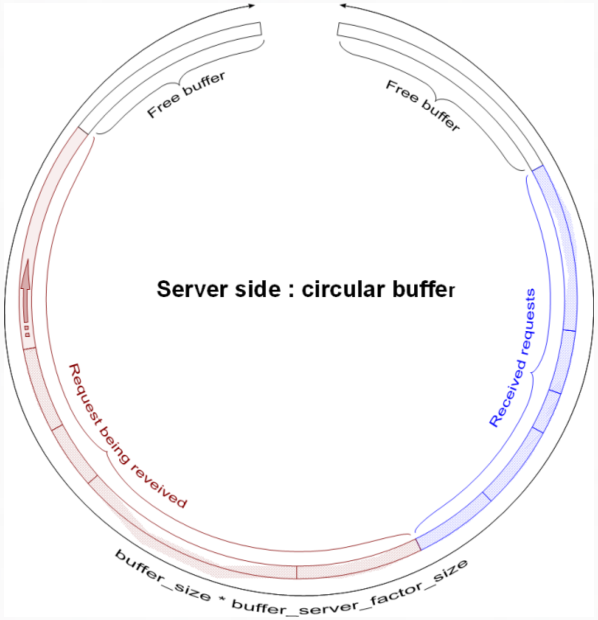
- Use buffer to smooth large peaks of data flow (monthly or daily output)
  - Several messages can be concatenated in one MPI call

- Client Side protocol : use double buffer
  - one for message transferring, one for buffering



Client side : double buffer

- Server side protocol
  - Using circular buffer
  - Received requests can be processed at the same time as a new request is transferring



Server side : circular buffer

IO Layer

- For now, output layer uses only NETCDF4/HDF5 parallel library.
  - optionally netcdf3 can be used, but without parallel support

- 2 modes are possible : "one_file" or "multiple_file"

- "Multiple_file" mode
  - One per XIOS servers
  - rebuilding phase is needed at post-processing

- "One_file" mode
  - All XIOS servers write simultaneously in a single file
  - Uses MPI/IO to aggregate file system bandwidth

- Achieving good performance with netcdf4/hdf5/MPI_IO layer is very challenging
  - strong file system dependence
  - a lot of recipes to avoid very bad performance, a lot of work done.

- XIOS embeds an improved netcdf4 parallel library for improved performance
  - netcdf compilation is managed by XIOS.

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

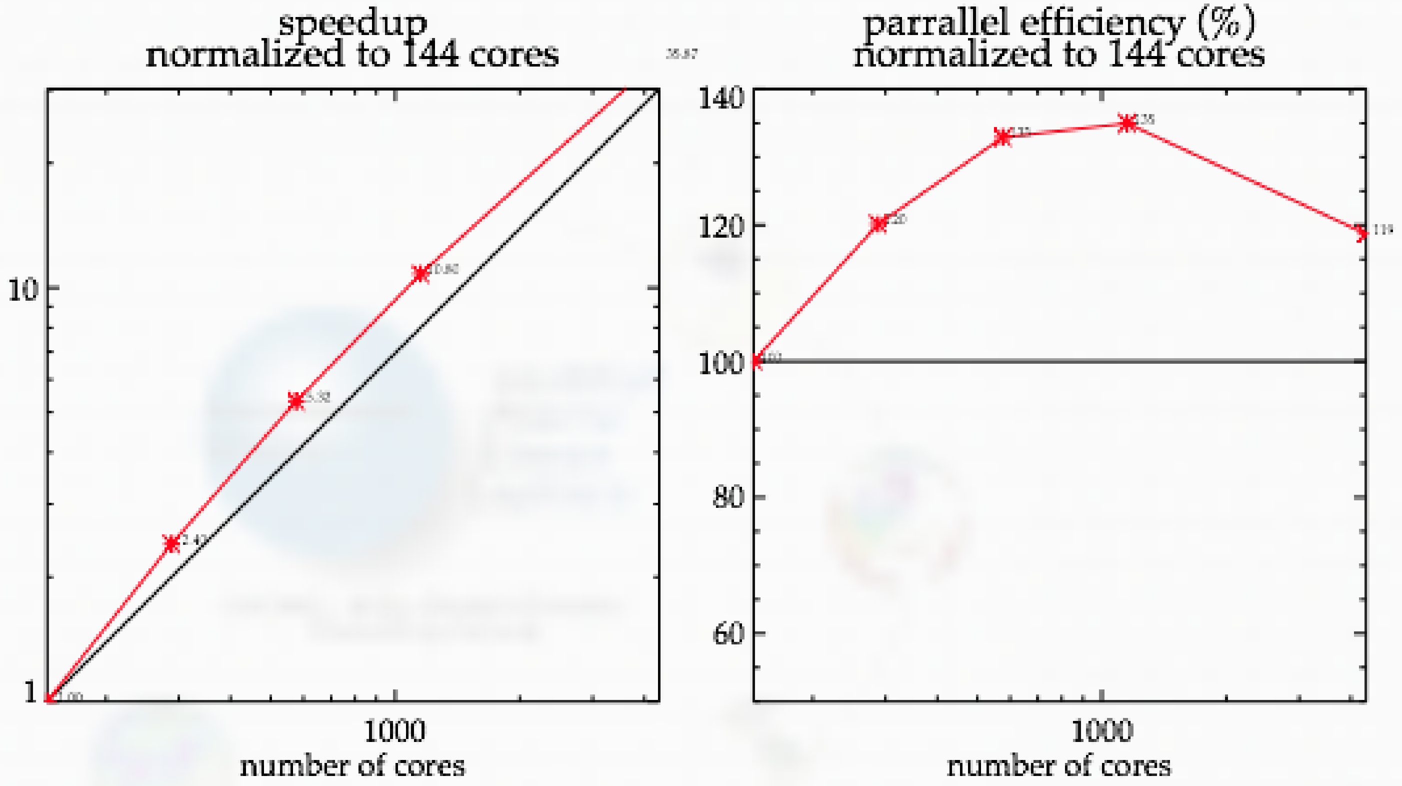
- Very huge configuration : 1/12th degrees global
  - GYRE 144 : 4322x2882x31 , up to 8160 cores
  - 6 day simulation (2880 time steps), hourly means output : 300s run, ~1.1 Tb
  - **3.6 Gb/s, 13 Tb/hour, 312 Tb/day, 9.4 Pb/month (real time)**
- File system capability : Lustre 150 Gb/s (global) theoretically
- In practice with an optimal MPI/IO simple parallel write test-case in a single file
  - must tune the number of OSTs used
  - peak ~ 20 Gb/s, average 10 Gb/s

- With NETCDF4/HDF5/MPI_IO layer on an ideal test case
  - only MPI_IO call : ~ 8 Gb/s
  - whole < 5 Gb/s average

Multiple file mode

- Works fine, good scaling up to 8160 cores

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



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

One file mode

- More challenging, recents results...
  - Gyre 144, daily mean output
    - 8160 NEMO, 32 XIOS : works almost perfectly
      - **+1.5% for IO** < OS jitter
  - Gyre 144, 6 hourly mean output
    - 8160 NEMO, 32 XIOS
      - **+5% for IO**
  - Gyre 144, hourly mean output
    - 1024 NEMO, 16 XIOS
      - **+5% for IO**
    - 8160 NEMO, 128 XIOS
      - Extreme testcase, close to NEMO strong scalability limit.
      - Close to filesystem capability bandwidth.
      - **+ 15-20% for IO**
      - Netcdf bandwith could be improved ?
      - Maybe network jitter between NEMO MPI and client-server-lustre IO communication?