Best practices for parallel IO and MPI-IO hints

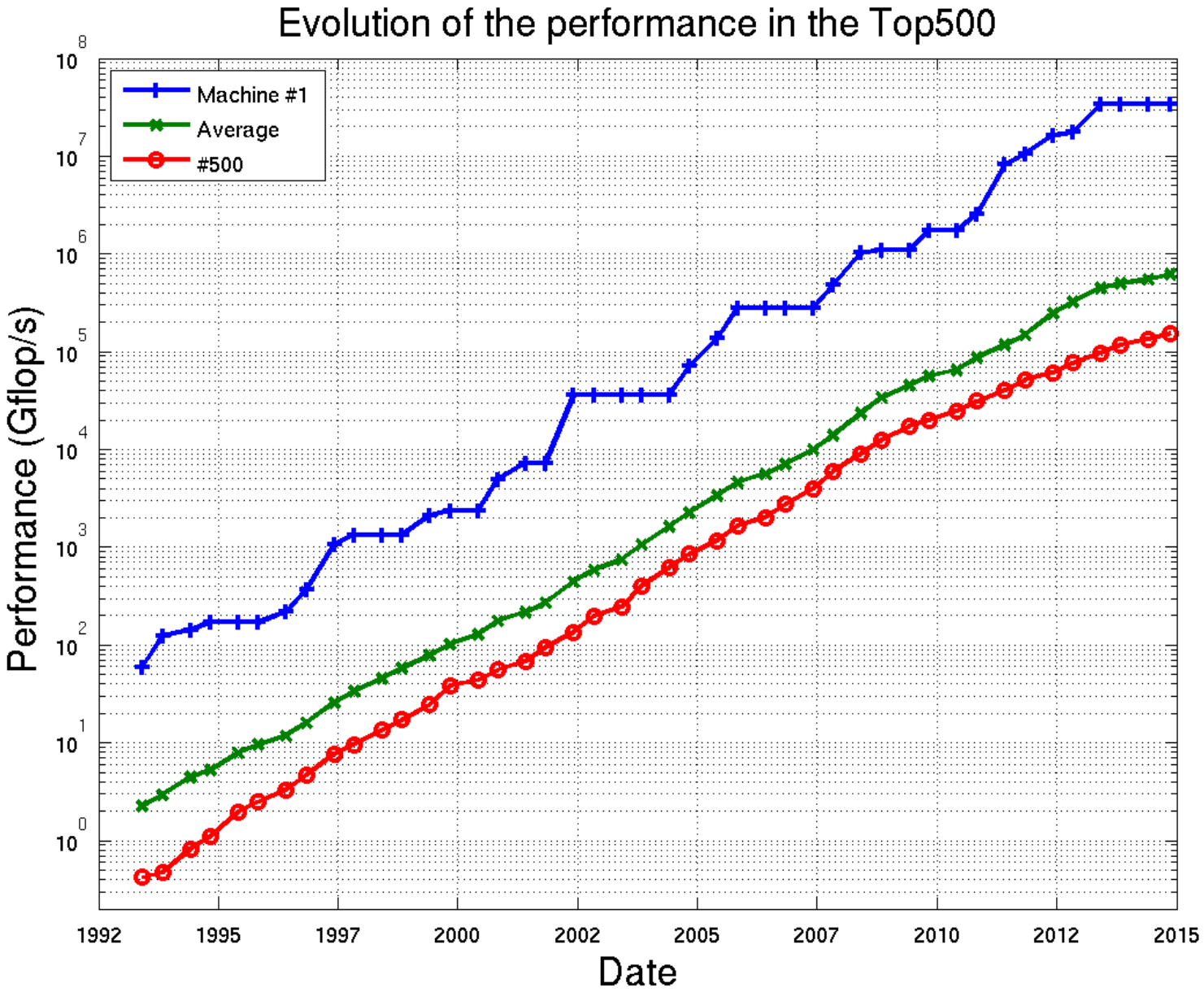
Philippe.Wautelet@idris.fr

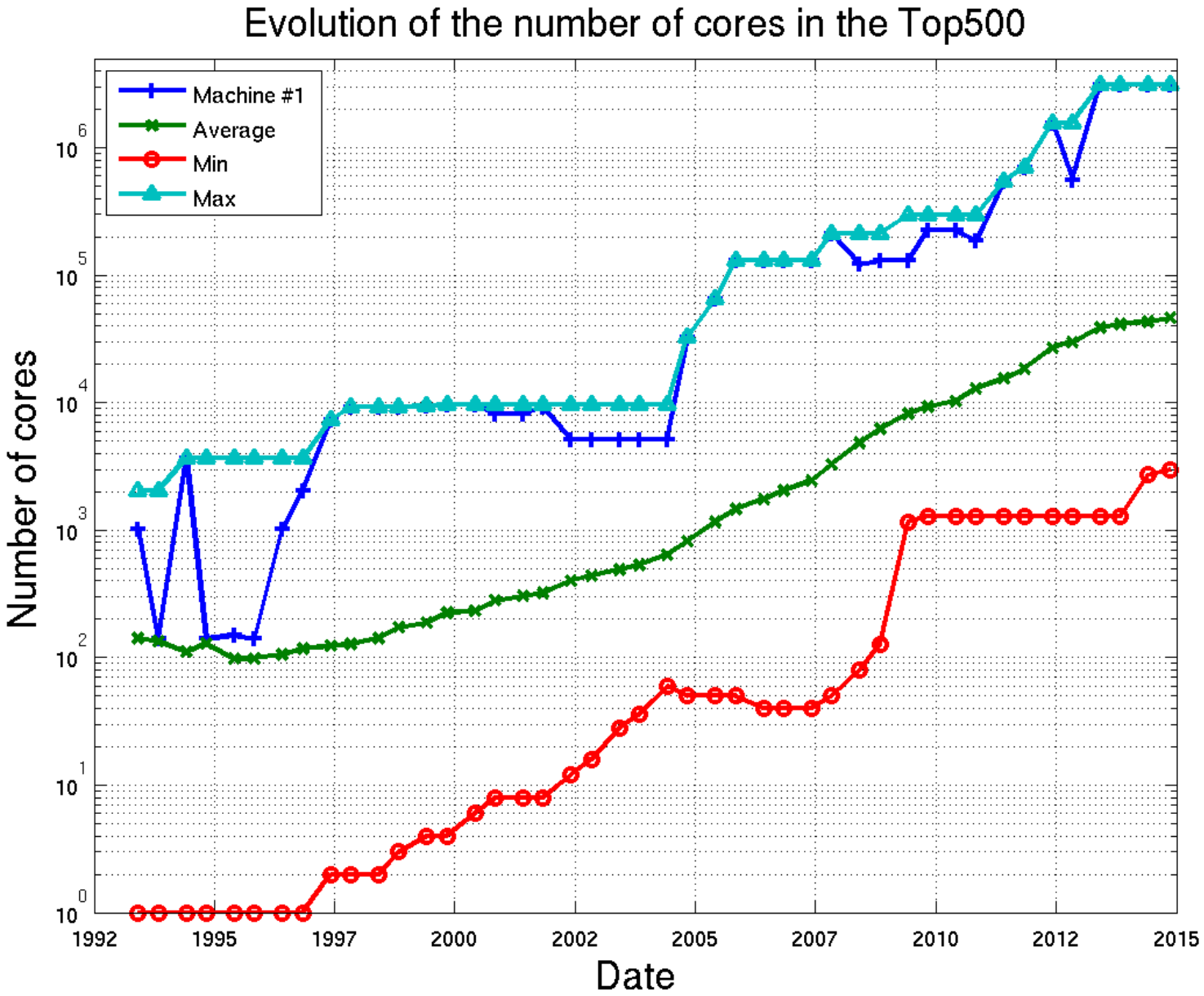
CNRS** - **IDRIS

PATC Training session
Parallel filesystems and parallel IO libraries
Maison de la Simulation / March 5th and 6th 2015

Plan

Introduction


Evolution of the performance in the Top500
Machine #1
Average
#500
Performance (Gflop/s)
Date
1992
1995
1997
2000
2002
2005
2007
2010
2012
2015


Evolution of the number of cores in the Top500
Machine #1
Average
Min
Max
Number of cores
Date
1992
1995
1997
2000
2002
2005
2007
2010
2012
2015

Technical evolution

- The computing power of supercomputers is doubling every year (faster than Moore's Law)
- The number of cores increases rapidly (massively parallel and many-cores architectures)
- The memory per core is constant and starts to decrease.
- The storage capacity is growing faster than the access speed (SSDs may change this)
- Disk throughput increases more slowly than the computing power
- I/O infrastructures becomes more complex (number of levels, performance tuning...)

Consequences

- The amount of data generated increases with the computing power (thankfully slower)
- At IDRIS, the quantity of stored data is doubling every 2 years and the number of files is multiplied by 1.5 every 2 years (slowing down)
- The standard one file per process approach generates more and more files resulting in saturation of metadata servers (too many simultaneous accesses and risks of crash)
- Less memory by core and more cores may reduce the average file size
- Too many files = saturation of the filesystems (maximum number of supported files and wasted space due to fixed block size)
- The time spent in I/O increases (especially for massively parallel applications)
- The pre-and post-processing steps are becoming heavier (need for parallelism ...)
- Managing millions of file is hard

MPI-IO hints

MPI-IO hints

Purposes

MPI-IO hints allow to direct optimisation by providing information such as file access patterns and file system specifics.

- I/O performance can be increased
- Use of system resources can be improved

MPI-IO hints

Usage

- MPI-IO hints are provided through *info* objects passed to `MPI_File_open`, `MPI_File_set_view` or `MPI_File_set_info`
- Hints are key - value pairs (for example, key is *romio_cb_write* and value is *enable*)
- `MPI_INFO_NULL` if no hints provided
- Unknown hints are ignored
- On some MPI implementations, hints can be set with environment variables (e.g. `MPICH_MPIIO_HINTS` and `MPICH_MPIIO_CB_ALIGN` for MPICH)

Procedure

- Create an *info* object with `MPI_Info_create`
- Set the hint(s) with `MPI_Info_set`
- Pass the *info* object to the I/O layer (through `MPI_File_open`, `MPI_File_set_view` or `MPI_File_set_info`)
- Free the *info* object with `MPI_Info_free` (can be freed as soon as passed)
- Do the I/O operations (`MPI_File_write_all`...)

C example

```c
MPI_File fh;
MPI_Info info;
MPI_Info_create(&info);

/* Enable the collective buffering optimisation */
MPI_Info_set(info, "romio_cb_write", "enable");

/* Set the striping unit to 4MiB */
MPI_Info_set(info, "striping_unit", "4194304");

MPI_File_open(MPI_COMM_WORLD, "hello",
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              info, &fh);

MPI_Info_free(&info);

MPI_File_write_all(fh, var, 1, MPI_INT, MPI_STATUS_IGNORE);
```

Fortran example

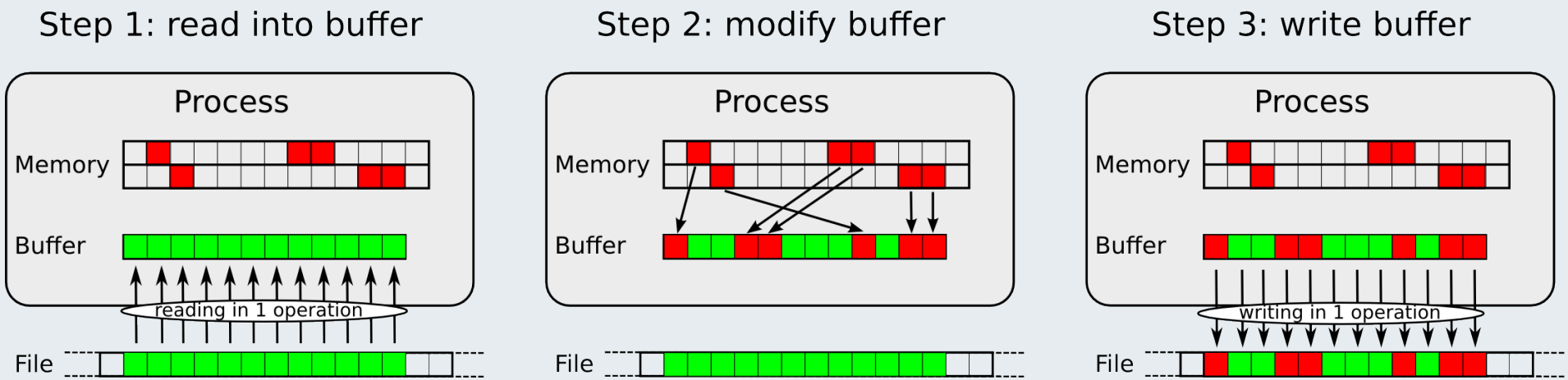
```
integer fh, ierr, info;
call MPI_Info_create(info,ierr)

! Enable the collective buffering optimisation
call MPI_Info_set(info, "romio_cb_write", "enable", ierr)

! Set the striping unit to 4MiB
call MPI_Info_set(info, "striping_unit", "4194304", ierr)

call MPI_File_open(MPI_COMM_WORLD, "hello",
                   MPI_MODE_WRONLY | MPI_MODE_CREATE,
                   info, fh, ierr)

call MPI_Info_free(info, ierr)

call MPI_File_write_all(fh, var, 1, MPI_INTEGER, MPI_STATUS_IGNORE,ierr)
```

Other MPI subroutines for hints

Call	*Usage*
`MPI_Info_dup`	To duplicate an *info* object
`MPI_Info_delete`	To delete a key - value pair
`MPI_Info_get`	To retrieve the value of a key
`MPI_File_get_info`	To get the hints values associated to a file
`MPI_Info_get_nkeys`	To get the number of defined keys
`MPI_Info_get_nthkey`	To extract a given key
`MPI_Info_get_valuelen`	To get the length of a key

Data sieving

- I/O performance suffers considerably when making many small I/O requests
- Access on small, non-contiguous regions of data can be optimized by grouping requests and using temporary buffers
- This optimisation is local to each process (non-collective operation)
- Warning: most filesystems perform very similar optimisations. Therefore, data sieving is often not useful and can even reduce performance (this has to be checked for each application/system combination).



Existing hints

Data sieving for reading

- A contiguous block of data is read into a local buffer
- Non-contiguous readings are made from the local buffer

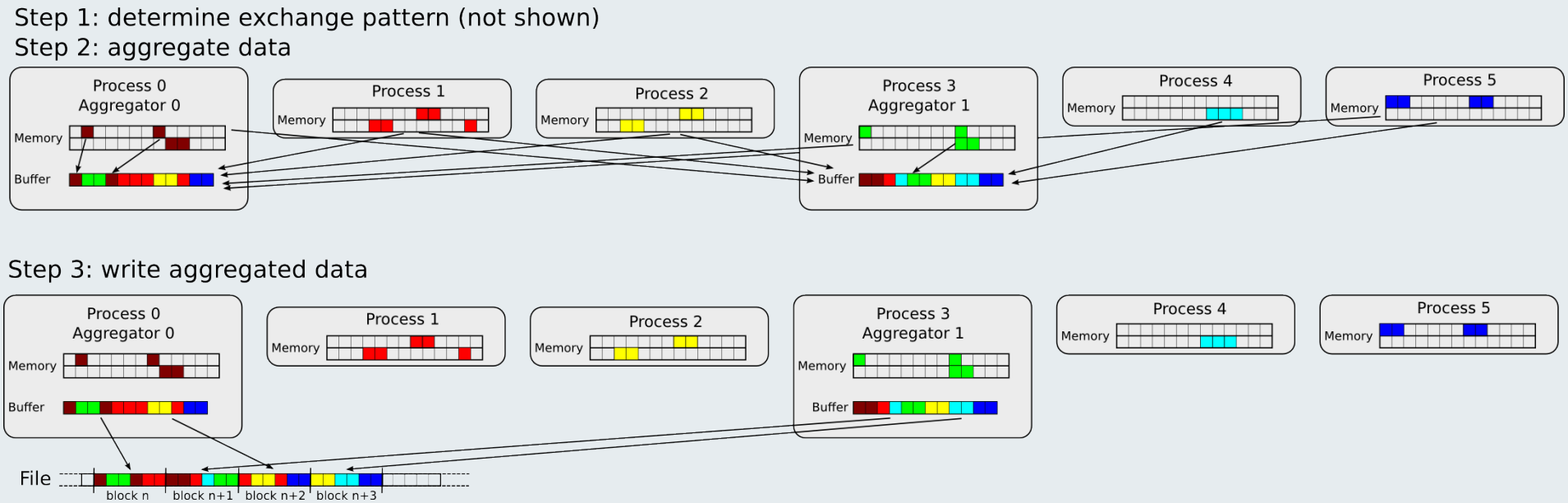
Data sieving for writing

- A contiguous block of data is read into a local buffer (read)
- Non-contiguous modifications are made in the local buffer (modify)
- The modified block is written back to the filesystem (write)

Collective buffering

Collective buffering, also called two-phase collective I/O, re-organises data across processes to match data layout in file.

- Mix of I/O and MPI communications to read or write data
- Communication phase to merge data from different processes into large chunks
- File accesses are done only by selected processes (called aggregators), the others communicates with them
- Large operations are split into multiple phases (to limit the size of the buffers and to overlap communications and I/O)



On Ada and Turing (IBM GPFS filesystem) and on CURIE (Lustre filesystem)

Existing hints and their usefulness for an application developer/user.

Hint	*Usefulness*	*Usage*
romio_cb_read	High	Enable or not collective buffering
romio_cb_write	High	Enable or not collective buffering
romio_cb_fr_types	Low	Tuning of collective buffering
romio_cb_fr_alignment	Low	Tuning of collective buffering
romio_cb_alltoall	Low	Tuning of collective buffering
romio_cb_pfr	Low	Tuning of collective buffering
romio_cb_ds_threshold	Low	Tuning of collective buffering
cb_buffer_size	Medium	Tuning of collective buffering
cb_nodes	Medium	Tuning of collective buffering
cb_config_list (not on Turing)	Medium	Tuning of collective buffering
romio_no_indep_rw	Low	Deferred open + only collective I/O
ind_rd_buffer_size	Low	Buffer size for data sieving
ind_wr_buffer_size	Low	Buffer size for data sieving
romio_ds_read	High	Enable or not data sieving
romio_ds_write	High	Enable or not data sieving

Most of the time, it is better to disable the data sieving optimisation because a similar one is already performed by the filesystem.

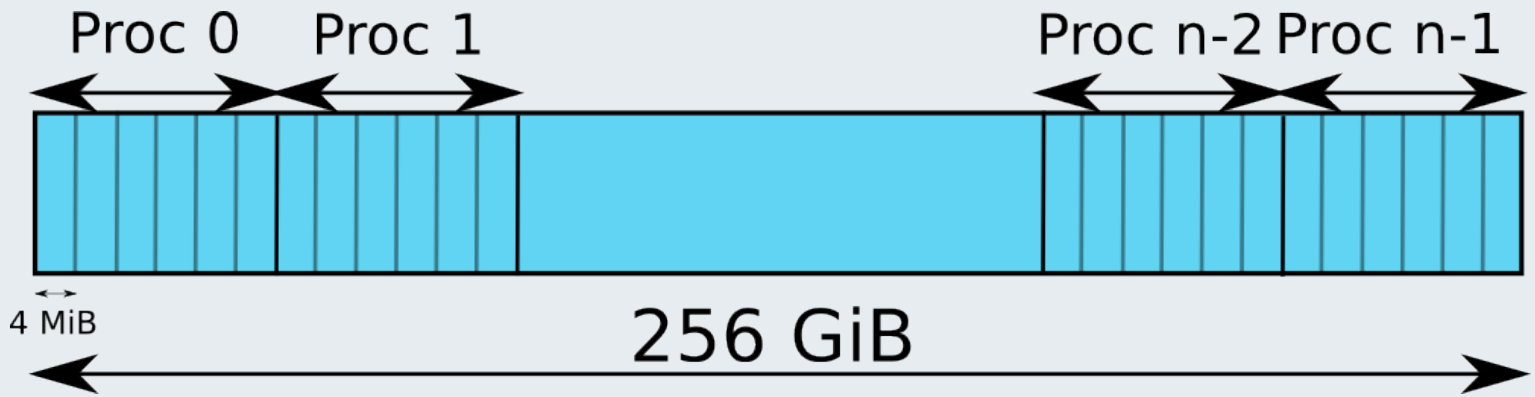
On CURIE (Lustre filesystem)

Existing hints and their usefulness for an application developer/user.
Same than Ada/Turing plus:

Hint	*Usefulness*	*Usage*
striping_unit	High	Size of each stripe
striping_factor	High	Number of OST ("disks") per file
romio_lustre_start_iodev	Low	Position of the first stripe
direct_read	Low	Direct I/O
direct_write	Low	Direct I/O
romio_lustre_co_ratio	Low	Max number of clients per OST in collective I/O
romio_lustre_coll_thresh	Low	Disable collective I/O if request bigger (except if set to 0)
romio_lustre_ds_in_coll	Low	Enable or not read-modify-write step in collective I/O

IOR overview

- IOR is a benchmark frequently used to evaluate parallel filesystems performance
- Handle separate POSIX, MPI-I/O, HDF5 and Parallel-NetCDF files
- Collective or individual mode
- Each process writes its part of the file of size *blocksize* by writing blocks of size *xfersize*
- Chosen sizes: total filesize: 256 GiB, $xfersize = 4MiB$ (corresponding to the GPFS blocksize)
- MPI-I/O hints (necessary for good collective performance):
  - on Ada : *romio_ds_write=disable*
  - on Turing : *romio_ds_write=disable* and *romio_cb_read=disable*



Hints on Ada: example

Writing tests of IOR on Ada with 128 processes and in collective mode.

Hint	MPI-I/O	HDF5	pNetCDF
romio_cb_write = automatic romio_ds_write = automatic	9893	1081	9592
romio_cb_write = enable romio_ds_write = automatic	429	459	465
romio_cb_write = automatic romio_ds_write = disable	**10824**	**8295**	**9772**

Values are in MiB/s. romio_cb_write = automatic and romio_ds_write = automatic are the default values.

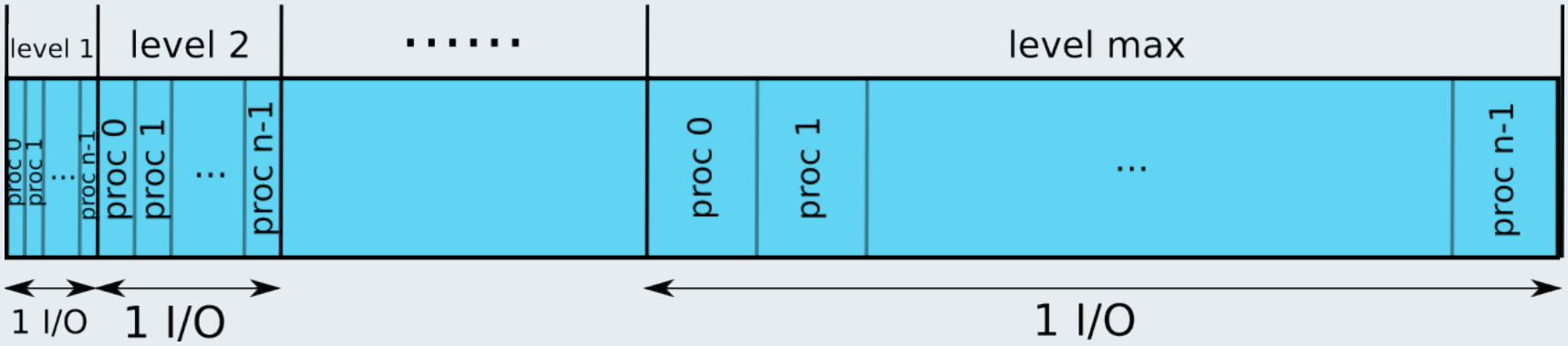
Overview

- RAMSES is an astrophysics application originally developed by Romain Teyssier (CEA) under CeCILL license (equivalent to GPL).
- Solves the Euler equations in the presence of self-gravity and cooling treated as source terms in the equations of time and energy.
- Adaptive mesh refinement (AMR method) with load balancing and dynamic memory defragmentation
- Multi-grid method and conjugate gradient for Poisson's equation
- Riemann solvers (Lax-Friedrich, HLLC, exact) for adiabatic gas dynamics
- Dynamic particles (without collisions) for the dark matter and the stars
- Star formation, supernovae...
- Code using MPI optimized by IDRIS (among others) and running on more than tens of thousands of processes on a regular basis

RAMSES is available at http://www.itp.uzh.ch/ teyssier/ramses/RAMSES.html

AMR file

- A set of identical parameters on all processes (14 integers, 22 double precisions, a small array of integers (10*nlevelmax) and a string)
- A set of different variables on each process (8 integers per process and 3 data structures (ncpu*nlevelmax integers per process)). The values on the different processes of a variable are arranged one behind another in the file.
- A set of large data structures (8 structures containing between 1 and 8 integers or double precision by grid cell) for a total of 156 bytes * number of cells. They are structured as follows:



Notes: grids (or levels) are of increasing sizes (factor 8 for the complete grids) and some processes may contain no values for some levels (depending on refinement).

RAMSES file structure

hydro file

- A set of identical parameters on all processes (5 integers, 1 double precision)
- A set of different variables on each process (2 data structures (ncpu*nlevelmax and 100*nlevelmax integers per process)). The values on the different processes of a variable are arranged one behind another in the file.
- A set of large data structures (5 structures containing eight double-precision values per cell) for a total of 320 bytes * number of cells. They are structured the same way as in AMR except that the values are grouped by 8 (so 8 times more values per level).

AMR file vs hydro file

- The AMR file is more complex and contains more variables
- The first 2 sections are much bigger in the AMR file (but still relatively small in absolute size)
- The hydro file is about 2 times larger (320 bytes per cell instead of 156 for the third section)
- The granularity of the entries is significantly larger in the hydro file

Hints on Ada: example

HDF5 writing tests (sedov3d 1024^3 256 processes)

Hint	*AMR*	*hydro*
romio_cb_write = automatic romio_ds_write = automatic	18.4s	15.1s
romio_cb_write = enable romio_ds_write = automatic	**16.9s**	**13.2s**
romio_cb_write = disable romio_ds_write = automatic	160.4s	77.3s
romio_cb_write = automatic romio_ds_write = enable	18.8s	15.9s
romio_cb_write = automatic romio_ds_write = disable	18.5s	16.0s

The first line corresponds to the default values.

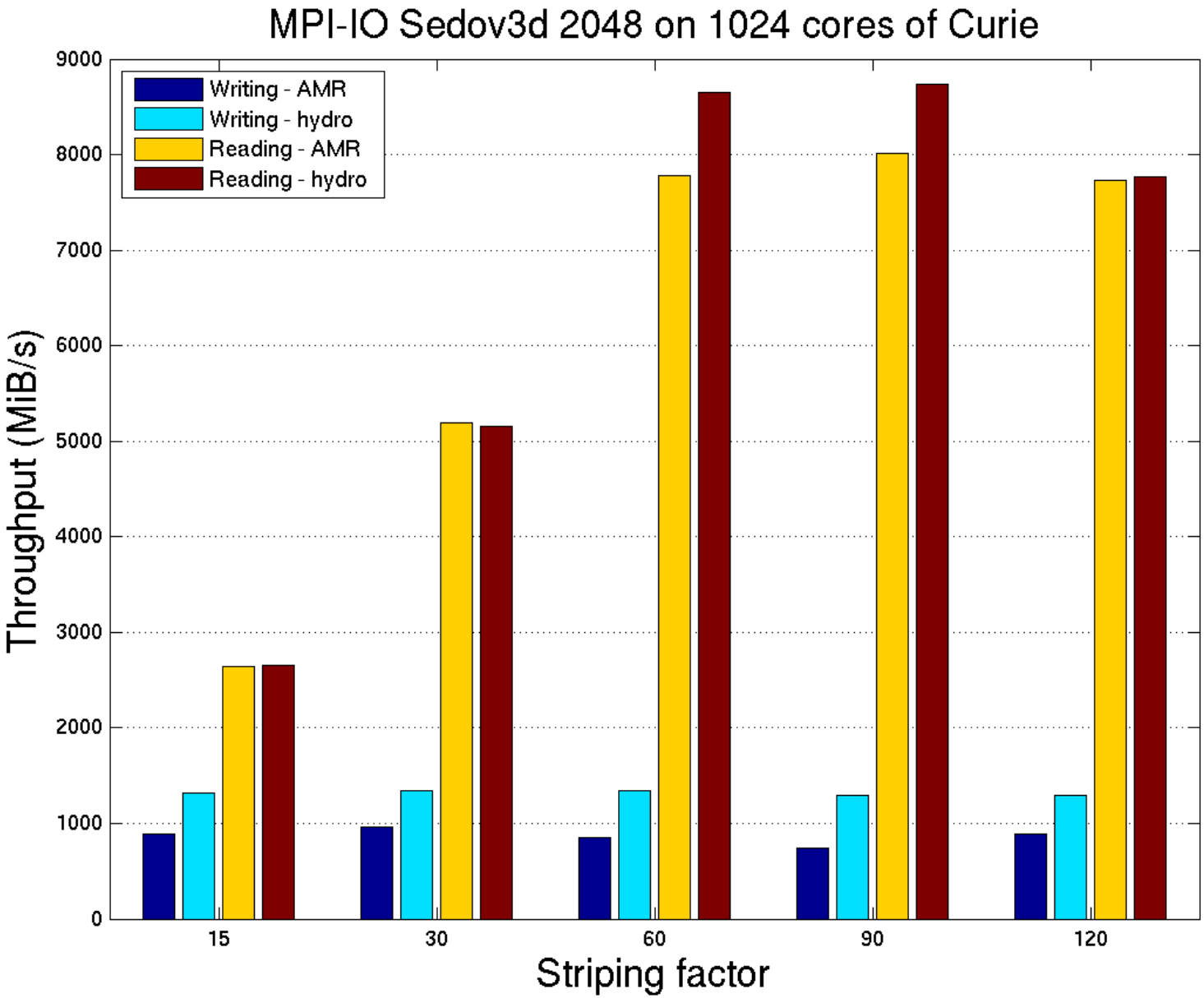
Hints on Blue Gene/Q: example

MPI-IO reading tests (sedov3d 1024^3 4096 processes (1 per core))

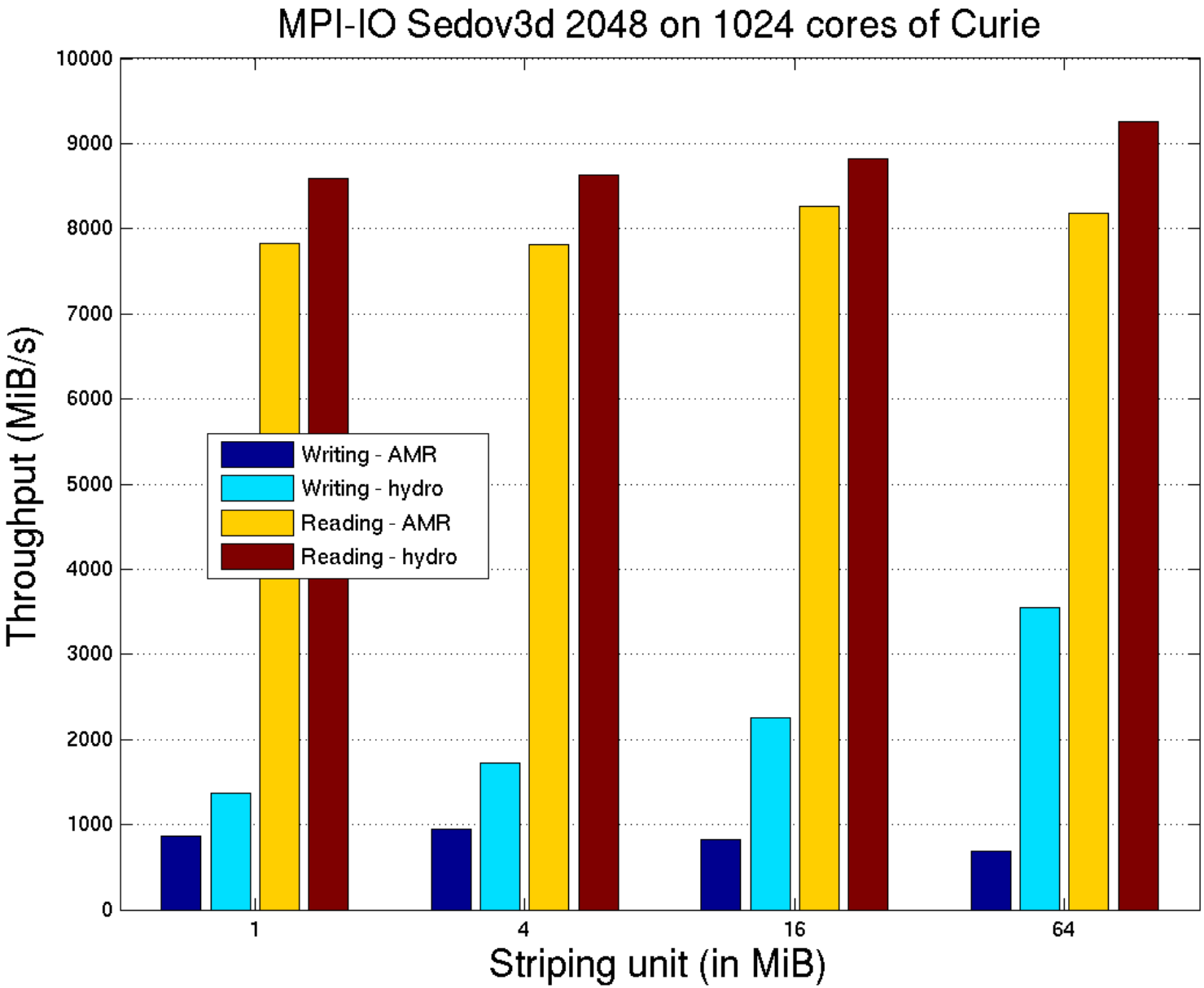
hint	*AMR*	*hydro*
romio_cb_read = enable romio_ds_read = automatic	**37.4s**	**18.1s**
romio_cb_read = disable romio_ds_read = automatic	1297.7s	177.9s
romio_cb_read = automatic romio_ds_read = automatic	1128.4s	149.0s
romio_cb_read = enable romio_ds_read = disable	42.2s	20.2s
romio_cb_read = enable romio_ds_read = enable	40.5s	18.8s

The first line corresponds to the default values.

Sedov3d 2048^3 on Curie: striping_factor effect



Sedov3d 2048^3 on Curie : striping_unit effect



Best practices for parallel I/O

3 **Best practices for parallel I/O**

- Why is my I/O performance bad?
- Do not write or read!
- General guidelines
- Some tricks or practices to try

Why is my I/O performance bad?

- Conflicts to access the same portion of the file (locks) and false sharing
- Contention with other I/O
- Random accesses
- I/O requests too small compared to the filesystem block size
- Read-Modify-Write effect
- Saturation of the metadata server(s)

Do not write or read!

- Write/read only what is necessary and when needed/useful
- Write/read as infrequently as possible (group small operations)
- Reduce accuracy (write in single precision, for example)
- Recalculate when it's faster

General guidelines

I/O needs and patterns are very different from one application to another. There are no universal rules. Beyond that, the following guidelines can prove successful:

- Use parallelism (multiple simultaneous accesses, parallel I/O libraries such as MPI-IO, HDF5, netCDF-4, Parallel-netCDF, SIONlib, ADIOS...)
- Limit the number of files (less metadata and easier to postprocess)
- Make large and contiguous requests
- Avoid small accesses
- Avoid non-contiguous accesses
- Avoid random accesses
- Prefer collective I/O to independent I/O (especially if the operations can be aggregated as single large contiguous requests)
- Use derived datatypes and file views to ease the MPI I/O collective work
- Try MPI I/O hints (especially the collective buffering optimisation; disabling data sieving is also very often a good idea; also useful for libraries based on MPI-IO)

Some tricks or practices to try

- Open files in the correct mode. If a file is only intended to be read, it must be opened in read-only mode because choosing the right mode allows the system to apply optimisations and to allocate only the necessary resources.
- Write/read arrays/data structures in one call rather than element per element. Not complying with this rule will have a significant negative impact on the I/O performance.
- Do not open and close files too frequently because it involves many system operations. The best way is to open the file the first time it is needed and to close it only if its use is not necessary for a long enough period of time.
- Limit the number of simultaneous open files because for each open file, the system must assign and manage some resources.
- Do make flushes (drain buffers) only if necessary. Flushes are expensive operations.

Some tricks or practices to try

- Separate procedures involving I/O from the rest of the source code for better readability and maintainability.
- Separate metadata from data. Metadata is anything that describes the data. This is usually the parameters of calculations, the sizes of arrays... It is often easier to separate files into a first part (header) containing the metadata followed by the data.
- Create files independent of the number of processes. This will make life much easier for post-processing and also for restarts with a different number of processes.
- Align accesses to the frontiers of the file system blocks and have only one process per data server (not easy).
- Create specialised processes (approach already followed by several groups; see the DAMARIS or XIOS libraries for example).
- Modify data structures (in file but also in memory).
- Use non-blocking MPI-I/O calls (not implemented/available on all systems).
- Use higher level libraries based on MPI-I/O (HDF5, netCDF-4, Parallel-netCDF, XIOS, ADIOS, SIONlib...).