I/O Optimization

Jan Thorbecke

jan@cray.com

THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

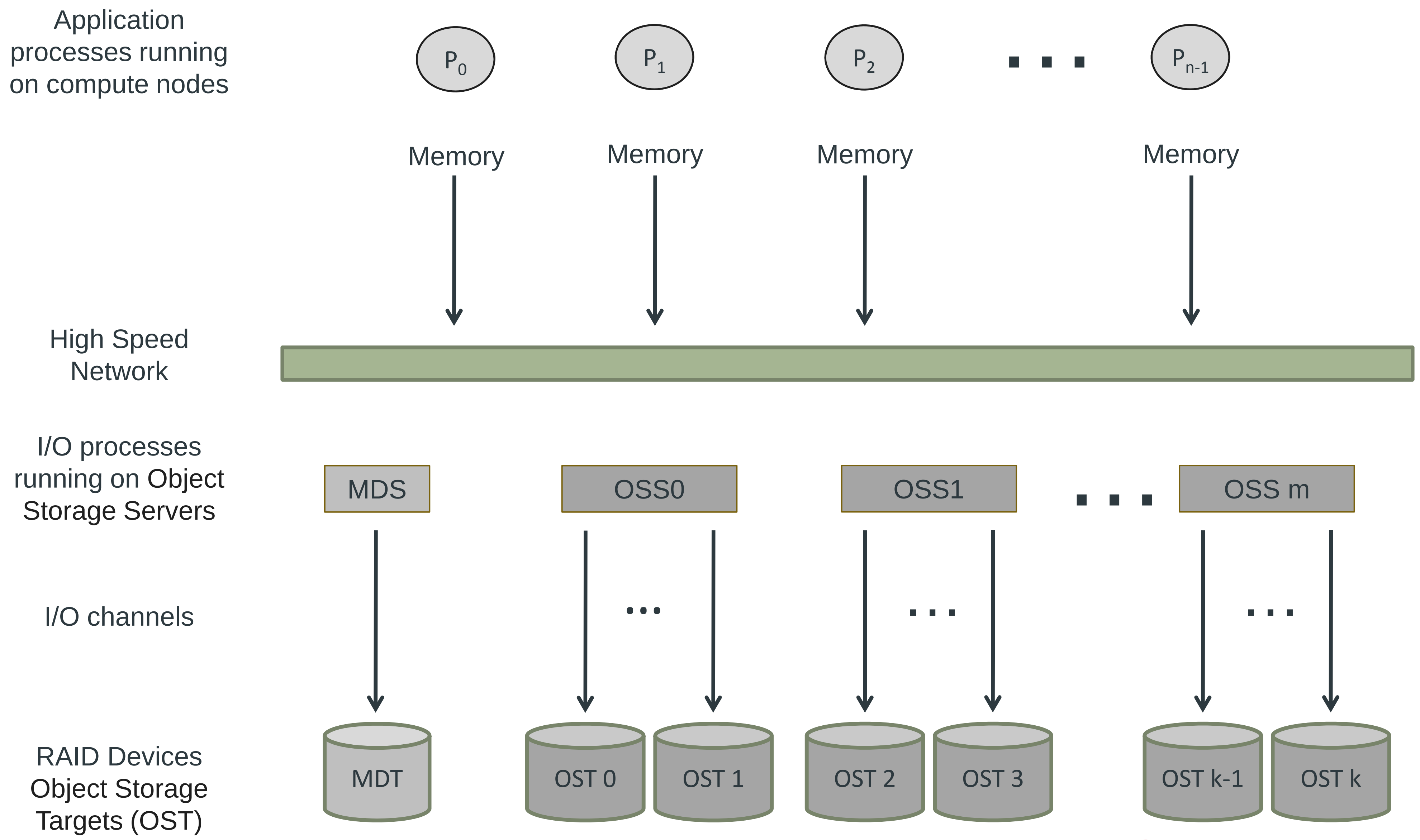
A supercomputer is a device for turning compute-bound problems into I/O-bound problems

Ken Batcher

Agenda

- Lustre for Users
  - Lustre, what is lustre and how can I use it
- Basic I/O strategies
  - How can parallel I/O be done
- Using MPI-IO
  - Performance and a few short code examples
- A simple but non trivial MPI-IO example

Basic Lustre Overview

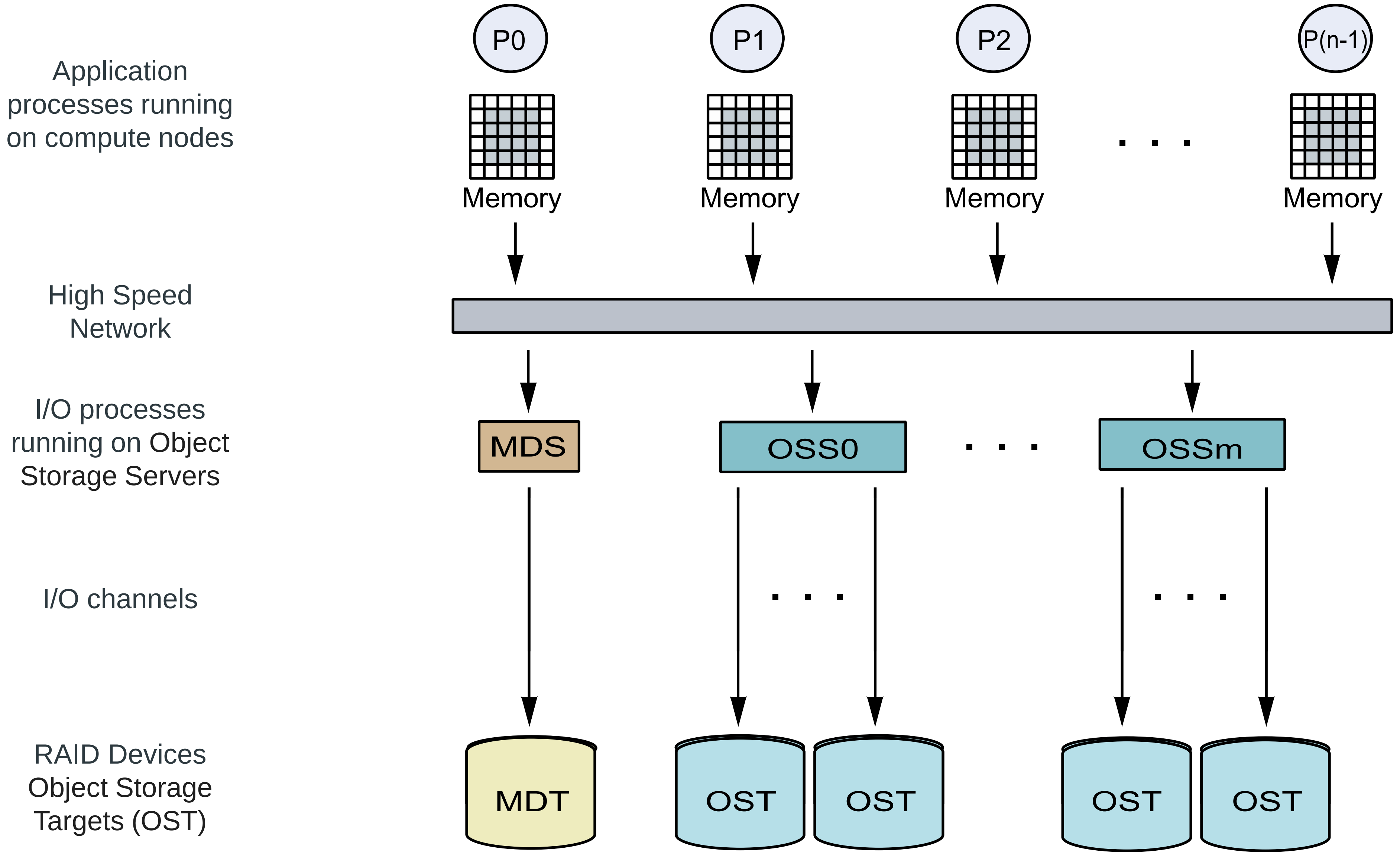


CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Basic Lustre Overview


Application processes running on compute nodes
P0
P1
P2
P(n-1)
Memory
Memory
Memory
. . .
Memory
High Speed Network
I/O processes running on Object Storage Servers
MDS
OSS0
. . .
OSSm
I/O channels
. . .
. . .
RAID Devices Object Storage Targets (OST)
MDT
OST
OST
OST
OST

THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

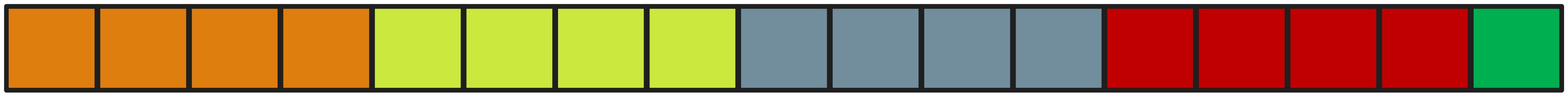
HP2C

Lustre Striping

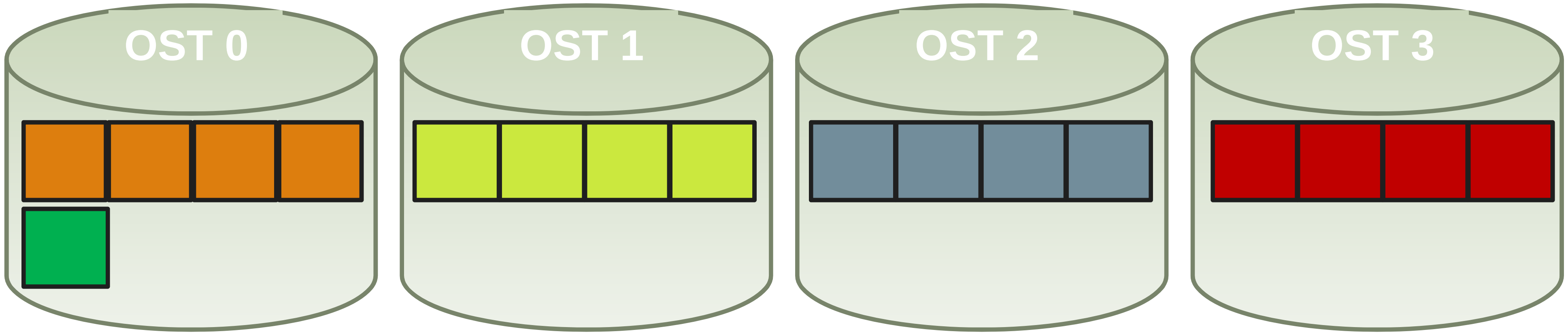
- The user can tell lustre how to stripe a file over the OST‘s.
- The number of bytes written to one OST before cycling to the next on is called the **„Stripe Size“**
- The number of OSTs across which the file is striped is the **„Stripe Count“**
  - The stripe count is limited by the number of OSTs on the filesystem you are using and has a current absolute maximum of 160
- The **„Stripe Index“** is the starting OST of the file
  - You can select the starting index, the others are selected by the SW
- You control the striping by the **„lfs“** command
- Your application does not directly reference OSTs or physical I/O blocks

Striping : Logical and Physical View of a File

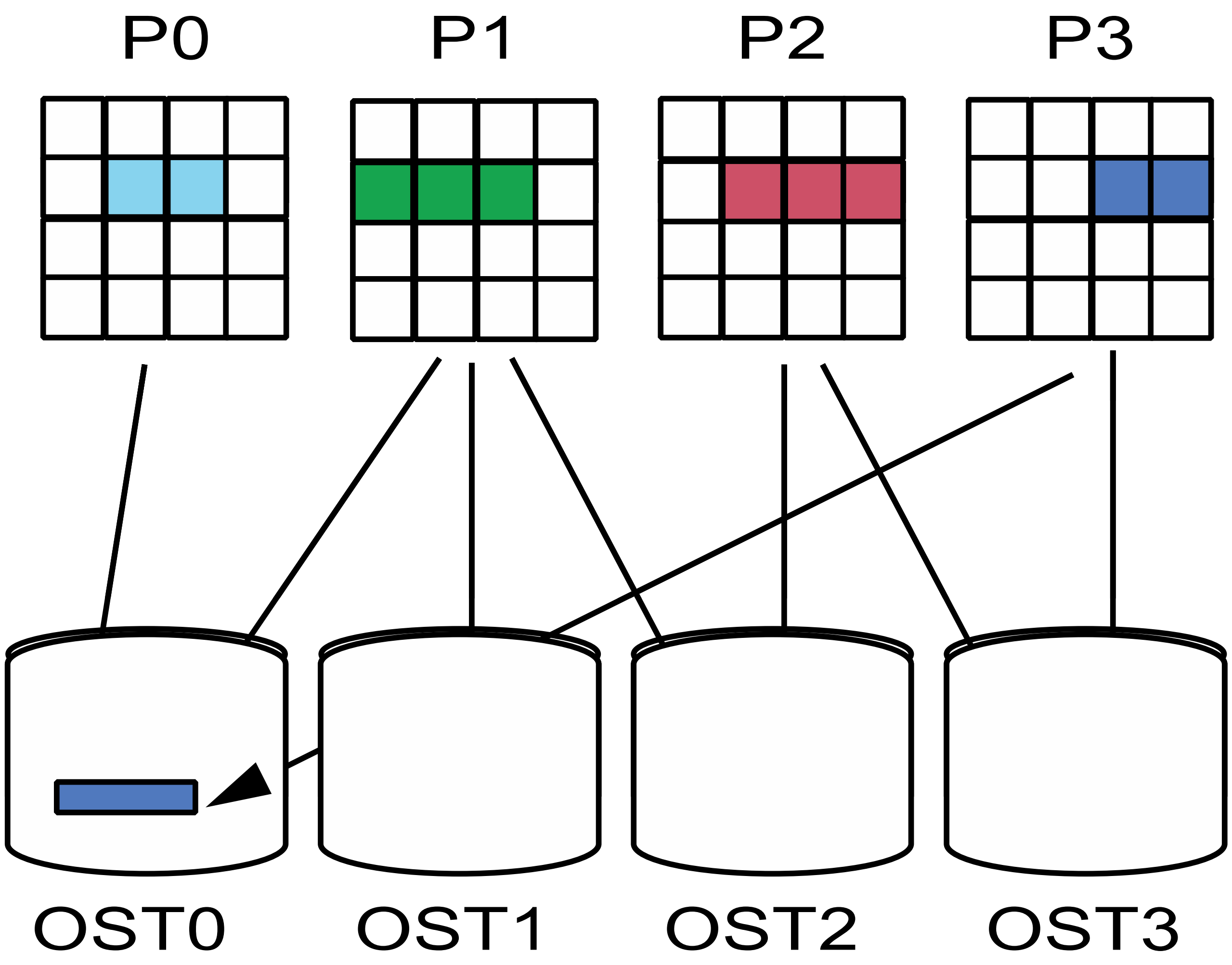
- Logically, a file is a linear sequence of bytes :



- Physically, a file consists of data distributed across OSTs.



Physical view of striping



Setting the stripe values

- „lfs setstripe“ is used to set the stripe information for a file or directory:

```
stefan@kaibab:~> lfs
lfs > help setstripe
setstripe: Create a new file with a specific striping pattern or
set the default striping pattern on an existing directory or
delete the default striping pattern from an existing directory
usage: setstripe [--size|-s stripe_size] [--offset|-o start_ost]
                 [--count|-c stripe_count] [--pool|-p <pool>]
                 <dir|filename>
       or
       setstripe -d <dir>   (to delete default striping)
        stripe_size:  Number of bytes on each OST (0 filesystem default)
                      Can be specified with k, m or g (in KB, MB and GB
                      respectively)
        start_ost:    OST index of first stripe (-1 filesystem default)
        stripe_count: Number of OSTs to stripe over (0 default, -1 all)
        pool:         Name of OST pool
lfs > quit
```

- The striping info for a file is set when the file is created. It cannot be changed
- You should not change the default **stripe_index** value
  - This to prevent a single OST being ‚overused‘ and running out of space

Rules on how which striping values are used for a file

- The ‘root’ filesystem has a default setting.
- A file/directory will inherit the setting of the directory it is created in
- You can change the setting of directory any time
  - This will only have an effect on new files, old files does NOT change their value
- You can create an empty file with a different settings then the directory by using „lfs setstripe <filename> <your setting>“ (think „touch“)
- You can create a file with specific striping values from your application using MPI-IO (coming up later)
- If you want to change the lustre settings on an existing file you have to copy it :

```
lfs setstripe <your settings> newfile
cp oldfile newfile
rm oldfile
mv newfile oldfile
```

Available Lustre filesystems and their basic information

- To check for available lustre filesystems, you do **lfs df –h**.

```
stefan66@emil-login2:~> lfs df -h
UUID     bytes Used Available Use% Mounted on
lustrefs-MDT0000_UUID 1.4T 655.5M 1.3T 0%  /mnt/lustre_server[MDT:0]
lustrefs-OST0000_UUID 3.6T 658.7G 2.8T 17% /mnt/lustre_server[OST:0]
lustrefs-OST0001 UUID 3.6T 717.4G 2.7T 19% /mnt/lustre server[OST:1]
lustrefs-OST0002_UUID 3.6T 712.0G 2.7T 19% /mnt/lustre_server[OST:2]
lustrefs-OST0003_UUID 3.6T 676.9G 2.7T 18% /mnt/lustre_server[OST:3]
filesystem summary:  14.3T 2.7T 10.9T 18%  /mnt/lustre_server

UUID bytes Used Available Use% Mounted on
ferlin-MDT0000_UUID 244.0G 534.3M 229.5G 0% /cfs/scratch[MDT:0]
ferlin-OST0000_UUID 8.7T 4.8T 3.5T 54% /cfs/scratch[OST:0]
ferlin-OST0001 UUID 8.7T 4.8T 3.5T 54% /cfs/scratch[OST:1]
ferlin-OST0002_UUID 8.7T 4.8T 3.5T 54% /cfs/scratch[OST:2]
ferlin-OST0003_UUID 8.7T 4.8T 3.5T 55% /cfs/scratch[OST:3]
ferlin-OST0004_UUID 8.7T 5.2T 3.1T 59% /cfs/scratch[OST:4]
filesystem summary: 43.6T 24.4T 17.1T 55% /cfs/scratch
stefan66@emil-login2:~>
```

CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Getting the stripe values

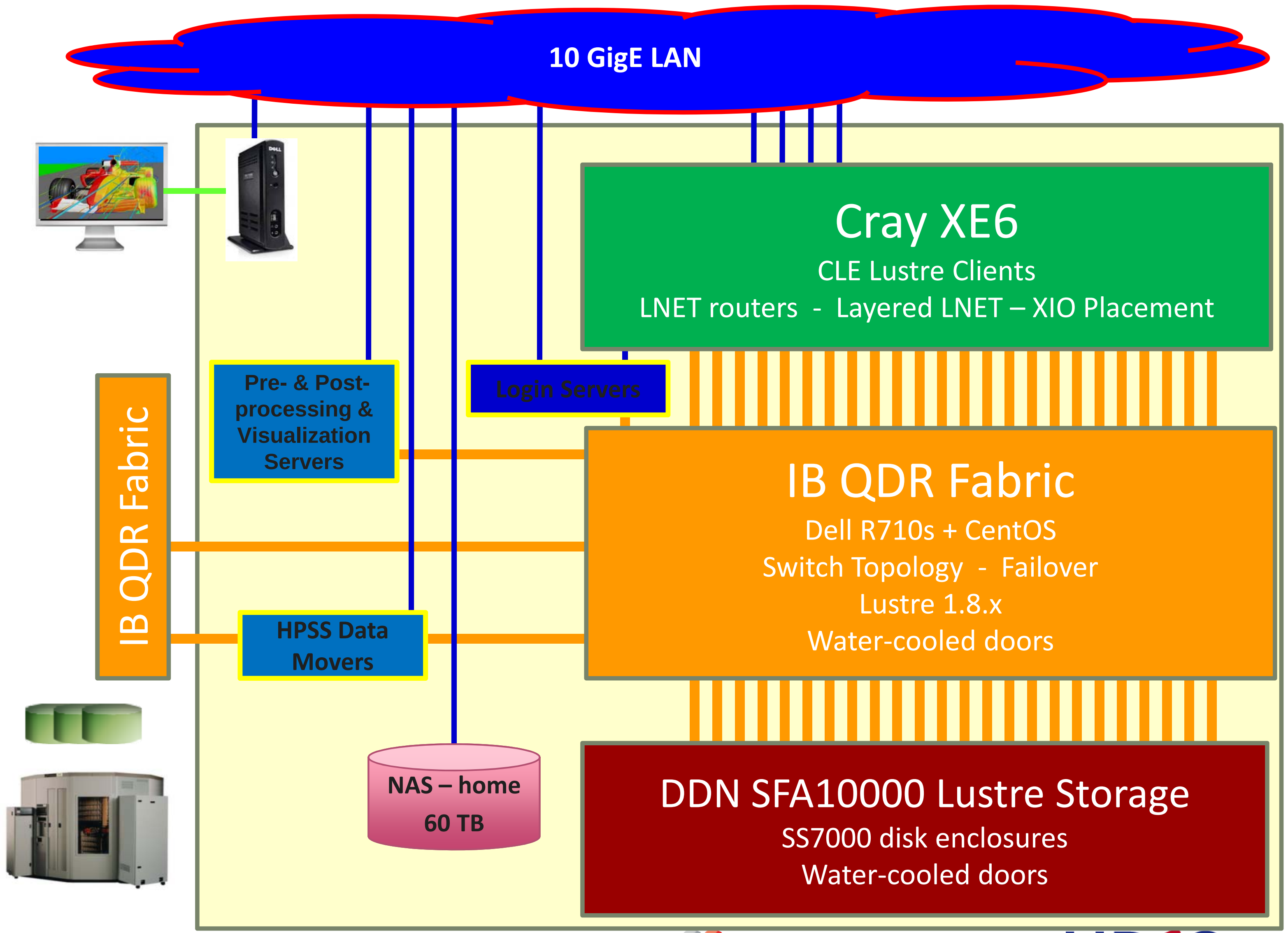
- **„lfs getstripe“** will return the striping information for a file or directory :

```
stefan@kaibab:/lus/scratch/stefan> touch delme
stefan@kaibab:/lus/scratch/stefan> lfs getstripe delme
delme
lmm_stripe_count:   12
lmm_stripe_size:    1048576
lmm stripe offset:  5
        obdidx           objid          objid            group
             5        29742704      0x1c5d670                0
             0        28810965      0x1b79ed5                0
            11        29259443      0x1be76b3                0
             9        28570631      0x1b3f407                0
             2        29447652      0x1c155e4                0
             1        30365044      0x1cf5574                0
             7        29045694      0x1bb33be                0
             8        30015537      0x1ca0031                0
             4        27747228      0x1a7639c                0
             6        27327312      0x1a0fb50                0
             3        29428807      0x1c10c47                0
            10        30076269      0x1caed6d                0
```

And lfs can more. Check the build-in help

```
stefan@kaibab:/lus/scratch/stefan> lfs
lfs > help
Available commands are:
        setstripe
        getstripe
        pool_list
        find
        check
        catinfo
        join
        osts
        df
        … (quota arguments removed)
        quota
        quotainv
        path2fid
        help
        exit
        quit
For more help type: help command-name
lfs >
```

Shared Lustre: Conceptual View



THE SUPERCOMPUTER COMPANY

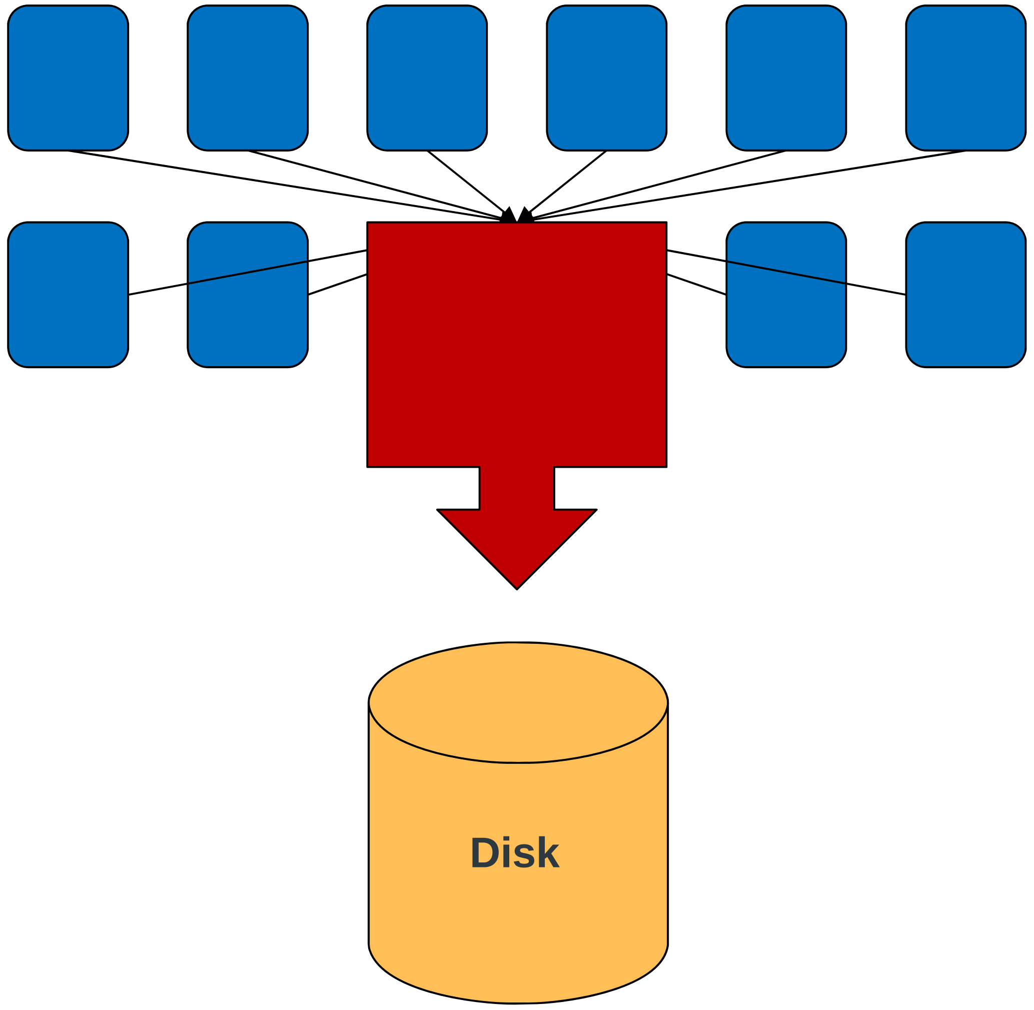
CSCS
Swiss National Supercomputing Centre

I/O Strategies

How can parallel I/O be done

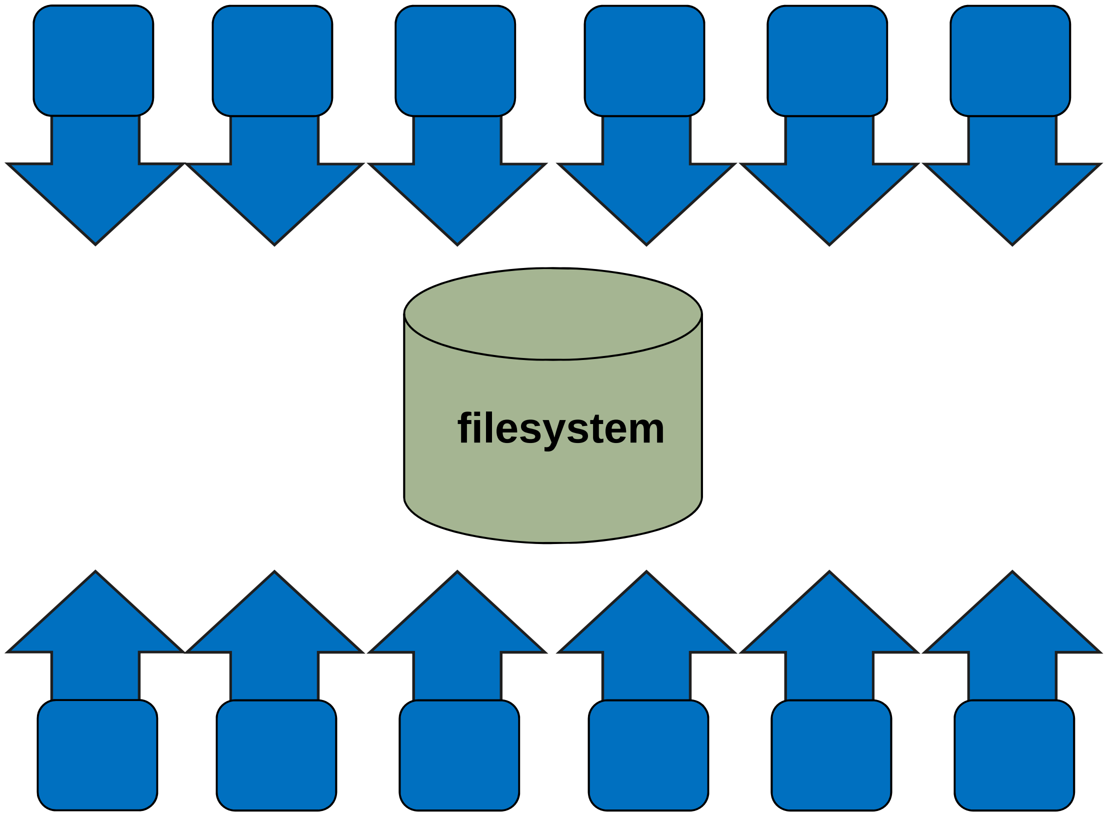
Spokesperson, basically serial I/O

- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.
- Easy to program
- Pattern does not scale.
  - Time increases linearly with amount of data.
  - Time increases with number of processes.
- Care has to be taken when doing the „all to one“-kind of communication at scale
- Can be used for a dedicated IO Server (not easy to program)



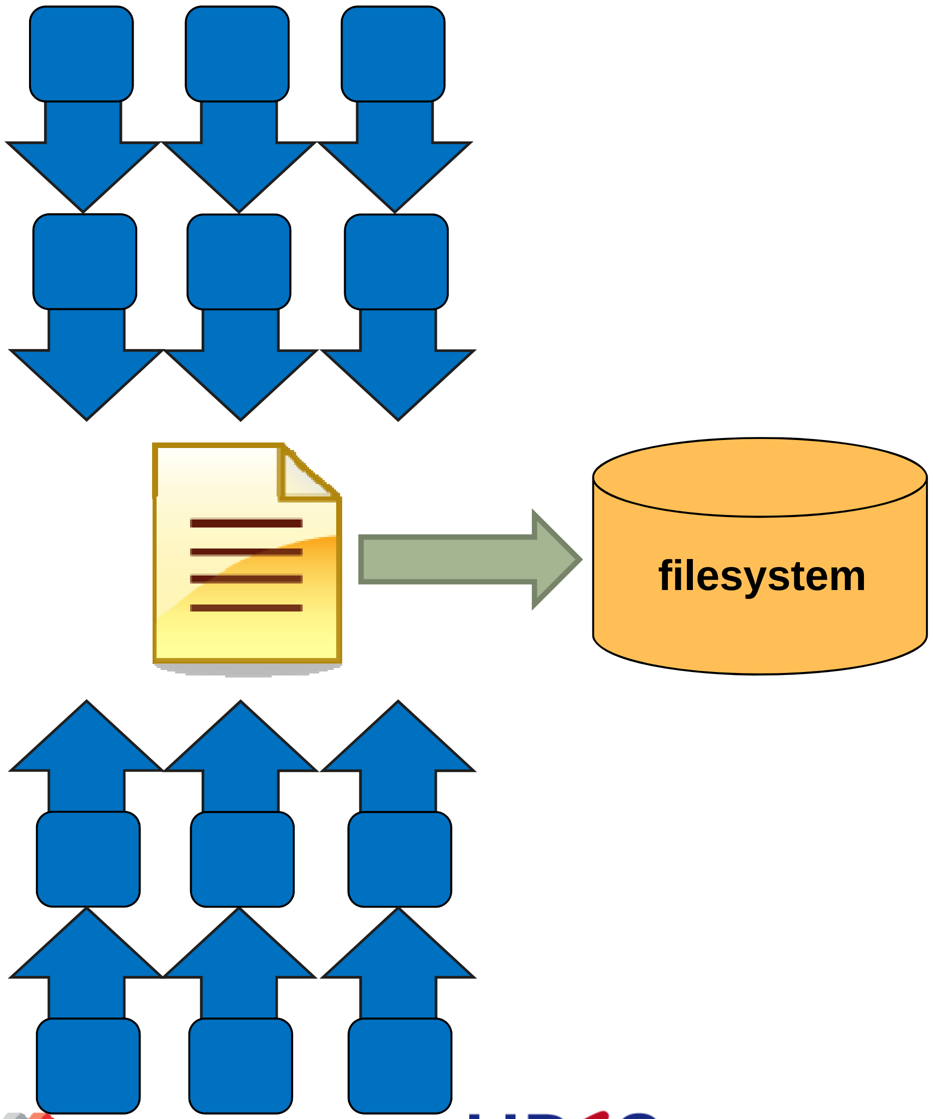
One file per process

- All processes perform I/O to individual files.
  - Limited by file system.
- Easy to program
- Pattern does not scale at large process counts.
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.



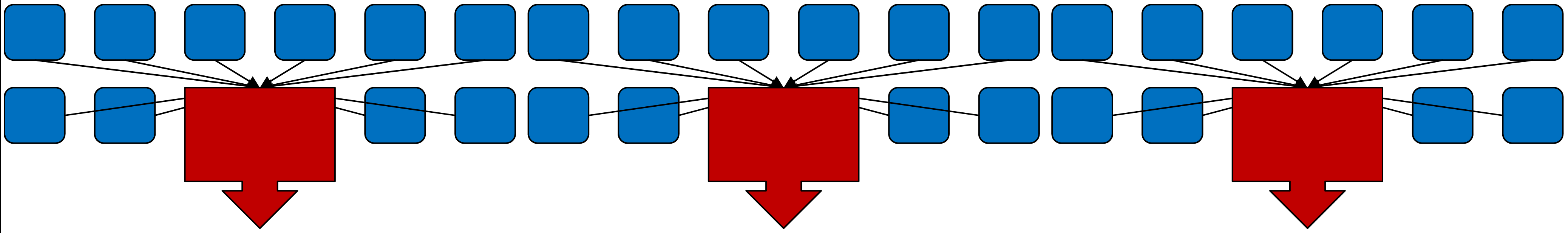
Shared File

- Each process performs I/O to a single file which is shared.
- Performance
  - Data layout within the shared file is very important.
  - At large process counts contention can build for file system resources.
- Programming language does not support it
  - C/C++ can work with fseek
  - No real Fortran standard



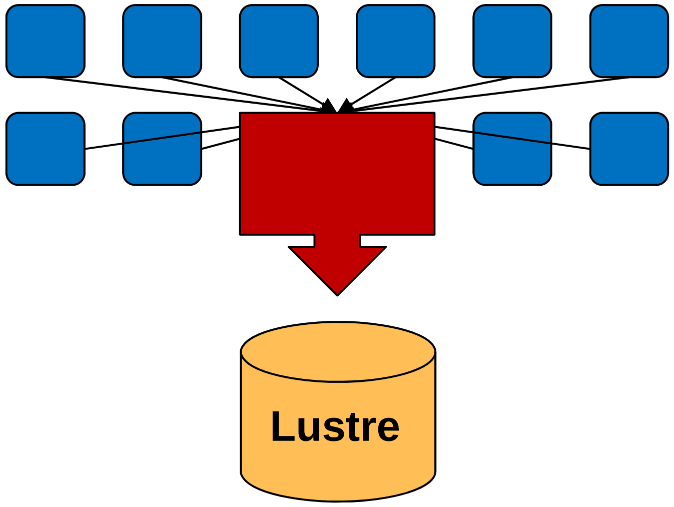
A little bit of all, using a subset of processes

- Aggregation to a processor group which processes the data.
  - Serializes I/O in group.
- I/O process may access independent files.
  - Limits the number of files accessed.
- Group of processes perform parallel I/O to a shared file.
  - Increases the number of shares to increase file system usage.
  - Decreases number of processes which access a shared file to decrease file system contention.

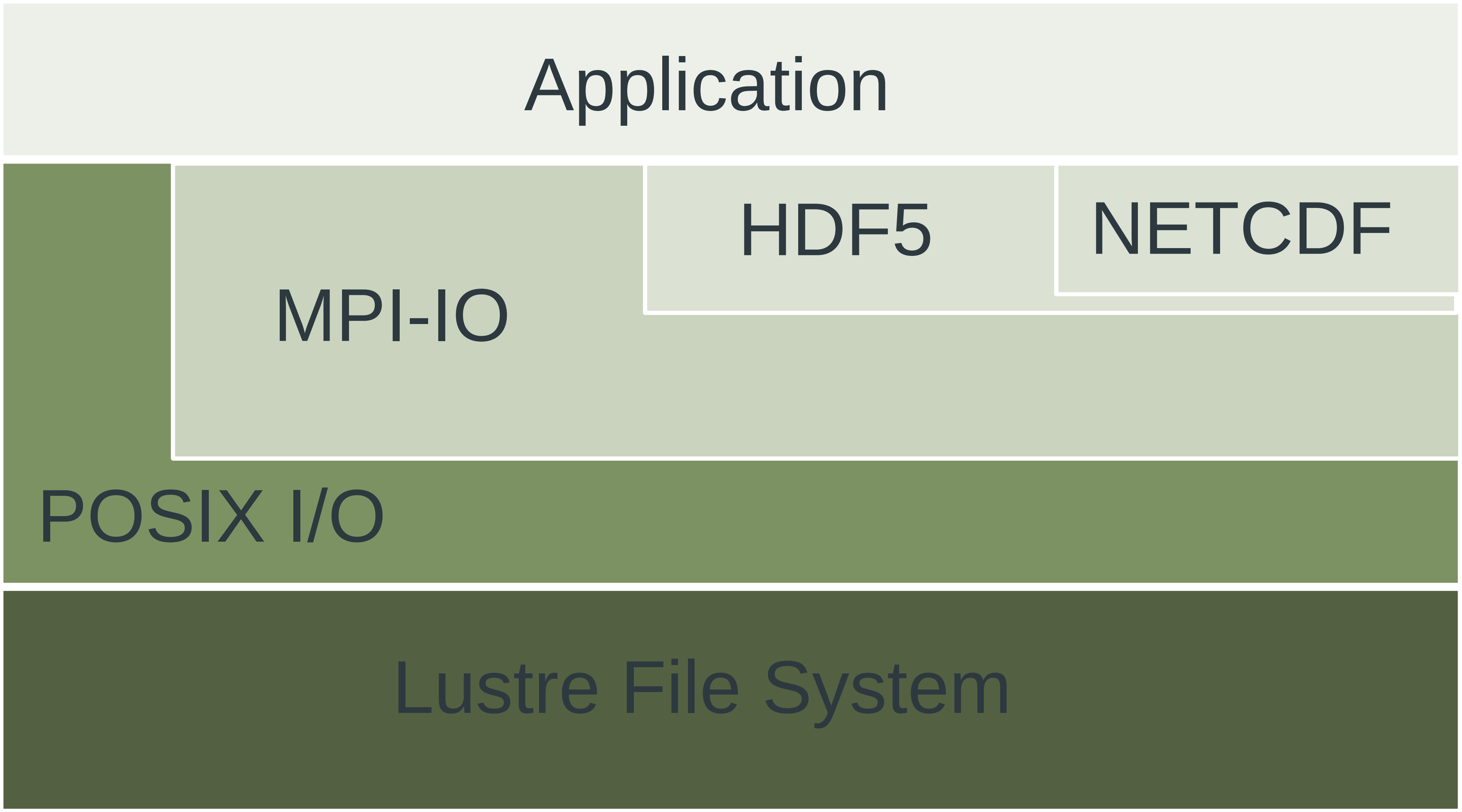


Special Case : Standard Output and Error

- Standard Output and Error streams are effectively serial I/O.
- All STDIN, STDOUT, and STDERR I/O serialize through aprun
- Disable debugging messages when running in production mode.
    - “Hello, I’m task 32000!”
    - “Task 64000, made it through loop.”
    - ...



CRAY IO Software stack



I/O Optimizations

‚outside‘ and ‚inside‘ your application

First step : Select best striping values

- Selecting the striping values will have an impact on the I/O performance of your application
- Rule of thumb :
  1. #files > #OSTs => Set stripe_count=1
     You will reduce the lustre contension and OST file locking this way and gain performance
  2. #files==1 => Set stripe_count=#OSTs
     Assuming you have more then 1 I/O client
  3. #files<#OSTs => Select stripe_count so that you use all OSTs
     Example : You have 8 OSTs and write 4 files at the same time, then select stripe_count=2

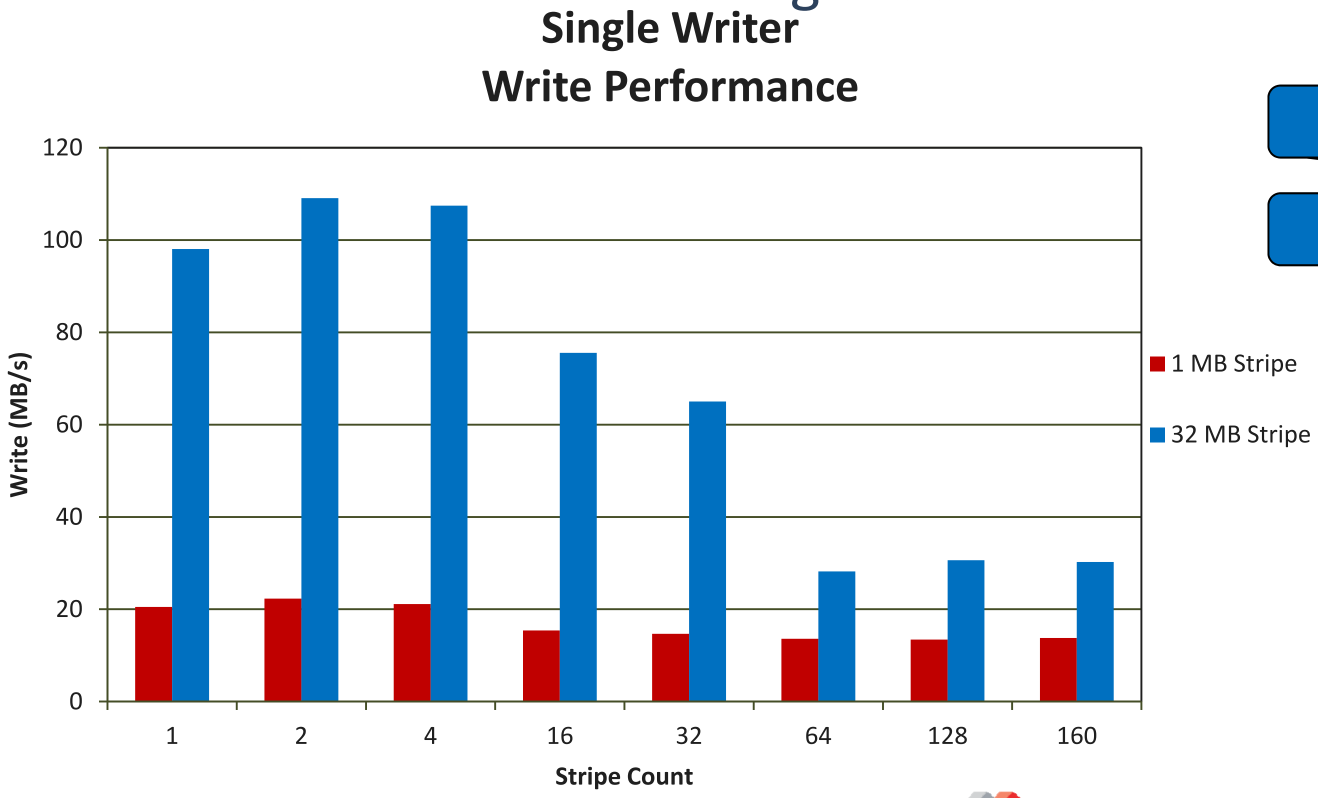
CRAY THE SUPERCOMPUTER COMPANY

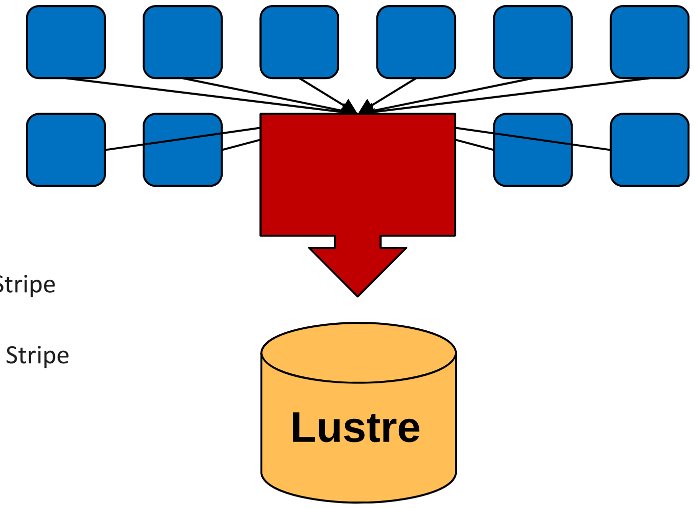
CSCS Swiss National Supercomputing Centre

HP2C

Case Study 1 : Spokesman

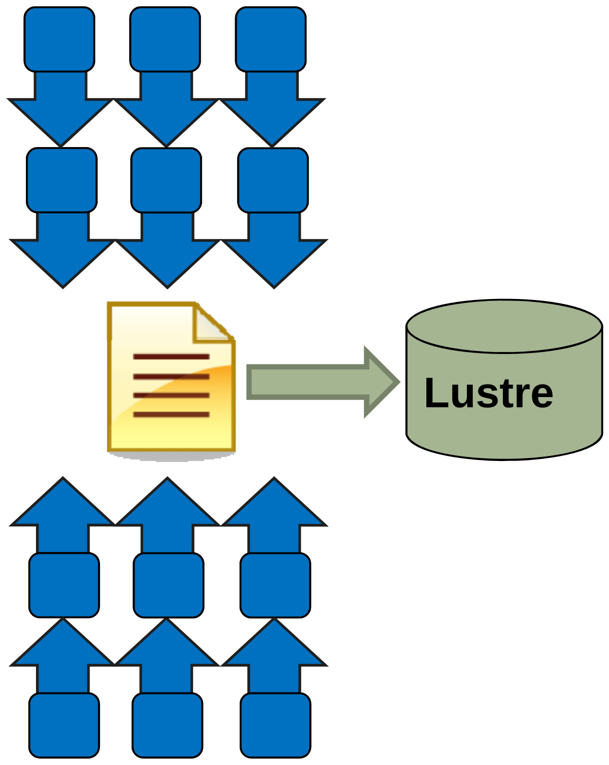
- 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance
  - Note : XE6 numbers might be better





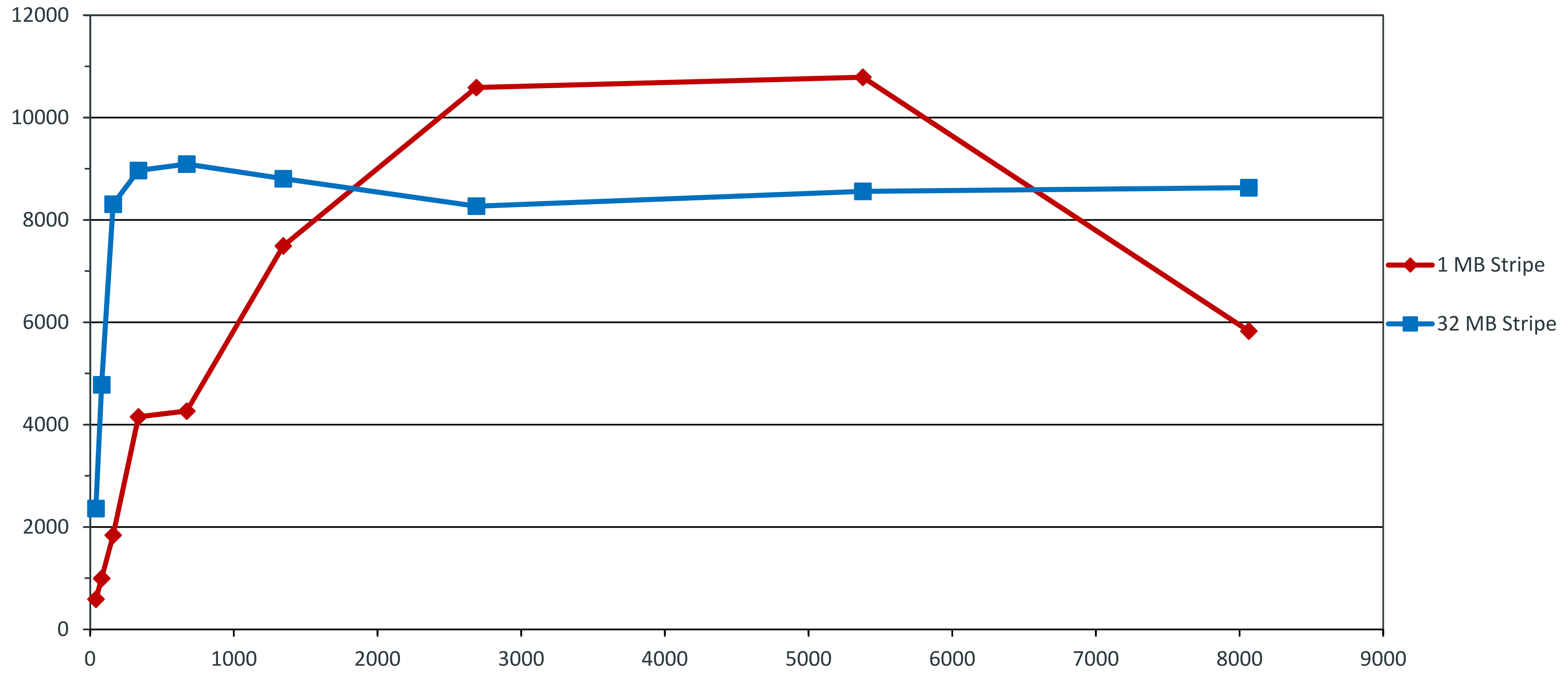
Case Study 2 : Parallel I/O into a single file

- A particular code both reads and writes a 377 GB file. Runs on 6000 cores.
  - Total I/O volume (reads and writes) is 850 GB.
  - Utilizes parallel HDF5
- Default Stripe settings:
  count =4, size=1M, index =-1.
  - 1800 s run time (~ 30 minutes)
- Stripe settings: count=-1, size=1M, index =-1.
  - 625 s run time (~ 10 minutes)
- Results
  - 66% decrease in run time.



Case Study 3 : Single File Per Process

- 128 MB per file and a 32 MB Transfer size, each file has a stripe_count of 1



Scaling the Met Office Unified Model on Cray XT

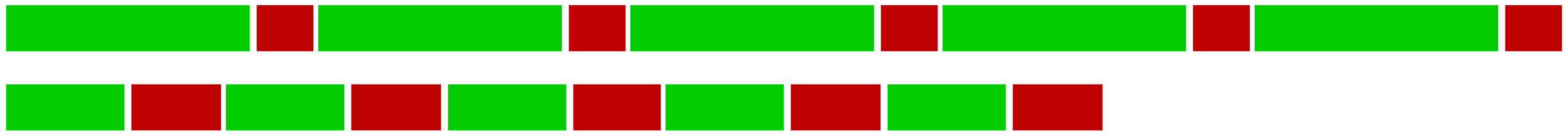
- The Met Office Unified Model (UM) is the numerical modelling system developed at the Met Office
  - It has been designed to allow different configurations of the same model to be used to produce weather forecasts and climate predictions
  - The system has been in continual development since 1990, taking advantage of steadily increasing supercomputer power, improved understanding of atmospheric processes, and an increasing range of observational data sources
  - The UM is highly versatile, capable of modelling a wide range of time and space scales including kilometre-scale mesoscale nowcasts, limited-area weather forecasts, global weather forecasts (including the stratosphere), seasonal foreasts, global and regional climate predictions as well as being run as part of an ensemble prediction system
  - The UM can be coupled to other models which represent different aspects of the Earth's environment that influence the weather and climate, such as the ocean and ocean waves, sea-ice, land surface, atmospheric chemistry and carbon cycle.
- Shown in the following is the N512L76 benchmark case
  - N512L76 is a 76 vertical level, 25km horizontal resolution (at mid-latitudes) global forecast model
  - The benchmark case is running 1 forecast day (normally run for 7 in operations)
  - The UM for this is running in a non-hydrostatic formulation, with Semi-Lagrangian advection and GCR solver. The grid used is an Arakawa C lat-long regular grid with Charney-Phillips vertical co-ordinate
- Acknowledgements
  - Paul Selwood – Met Office
  - Eckhard Tschirschnitz – Cray

UM Major Phases

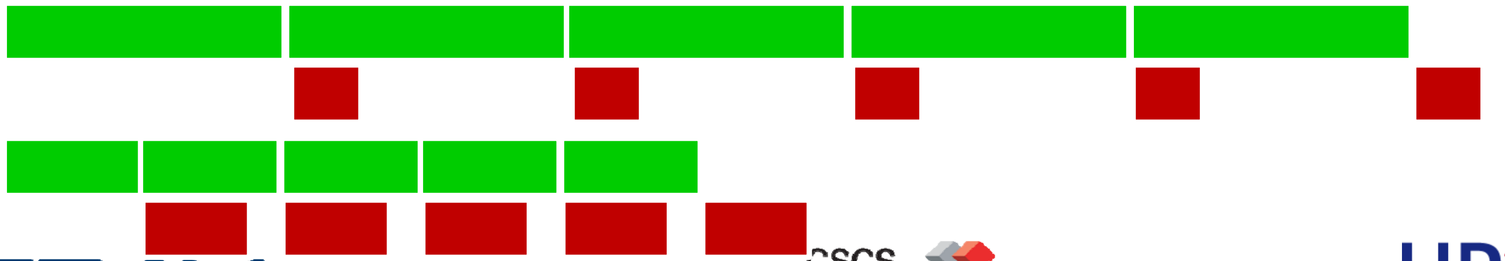
- Startup
  - Reading and distributing the initial input data
    - Goal: keep startup time constant as core count increases
- Simulation
  - Computation per timestep
    - Goal: optimize for cache based architecture
    - Goal: utilize hybrid MPI / OpenMP parallelism resulting in fewer and larger messages
    - Goal: optimize collective operations, which inherently are non-scaling
  - Collecting and writing the result data (frequency depending upon model)
    - Goal: fully hide behind computation through asynchronous I/O
- Shutdown
  - Collecting and writing the final Unified Model dump file
    - Goal: keep shutdown time constant as core count increases

UM per Timestep I/O

- UM already had implemented a definable number of asynchronous I/O servers
  - Each handling a certain number of files (Fortran I/O units)



- When doubling the number of cores, ideally compute time AND amount of I/O per core is reduced to half
- I/O time should scale – but it doesn‘t – how come?
  - The single I/O server per file becomes overwhelmed
  - Increasing number of smaller packets
  - I/O server collects data in a prescribed order, compute tasks wait for completion



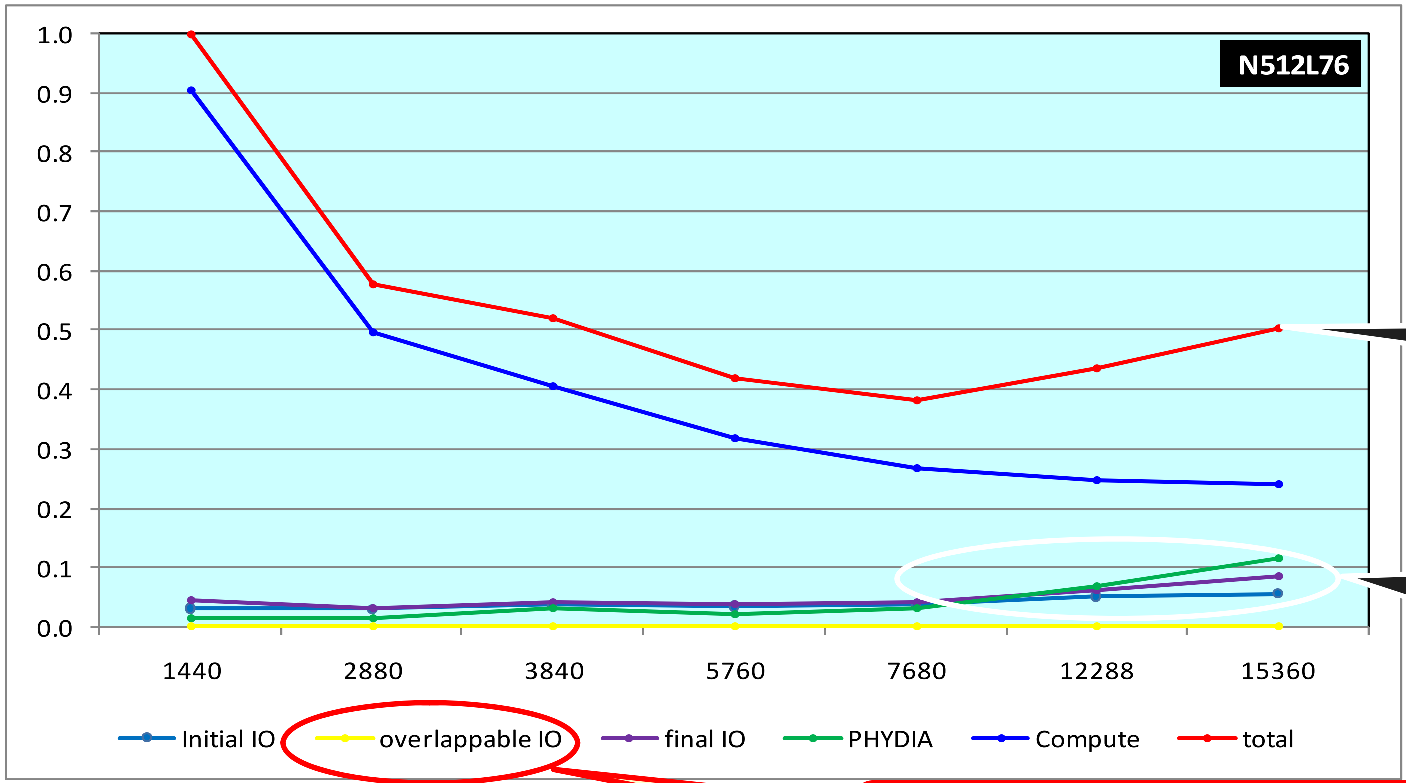
THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Impact of small Effects

- What works well at small core counts, may not at large core counts, and most likely will not at very large counts
  - Don‘t trust simple extrapolations
  - Fitting in between measurements is ok



Total time no longer dominated by (still scaling) compute

Init and shutdown not yet constant Diagnostic PHYDIA becoming important out of nowhere

Recurring IO is fully overlapped at all core counts

I/O Performance : To keep in mind

- There is no “One Size Fits All” solution to the I/O problem.
- Many I/O patterns work well for some range of parameters.
- Bottlenecks in performance can occur in many locations. (Application and/or File system)
- Going to extremes with an I/O pattern will typically lead to problems.
- I/O is a **shared** resource. Expect timing variation

MPI-IO

A simple MPI-IO program in C

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, ‘FILE’,
 MPI MODE RDONLY, MPI INFO NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

And now in Fortran using explicit offsets

```
use mpi ! or include 'mpif.h'
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset ! Note : might be integer*8

call MPI_FILE_OPEN(MPI_COMM_WORLD, ‘FILE’, &
 MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE
call MPI_FILE_READ_AT(fh, offset, buf, nints, MPI_INTEGER, status,
ierr)
call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count, 'integers'
call MPI_FILE_CLOSE(fh, ierr)
```

- The *_AT routines are thread save (seek+IO operation in one call)

Write instead of Read

- Use **MPI_File_write** or **MPI_File_write_at**
- Use **MPI_MODE_WRONLY** or **MPI_MODE_RDWR** as the flags to **MPI_File_open**
- If the file doesn't exist previously, the flag **MPI_MODE_CREATE must** be passed to **MPI_File_open**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+' or IOR in Fortran
- If not writing to a file, using MPI_MODE_RDONLY might have a performance benefit. Try it.

MPI_File_set_view

- MPI_File_set_view assigns regions of the file to separate processes
- Specified by a triplet (*displacement, etype, and filetype) passed to* MPI_File_set_view
  - *displacement = number of bytes to be skipped from the start of the* file
  - *etype = basic unit of data access (can be any basic or derived* datatype)
  - *filetype = specifies which portion of the file is visible to the process*
- Example :

```
MPI File fh;
for (i=0; i<BUFSIZE; i++) buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",MPI_MODE_CREATE |
 MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI File set view(fh, myrank * BUFSIZE * sizeof(int), MPI INT,
 MPI_INT, 'native', MPI_INFO_NULL);
MPI_File_write(fh, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

MPI_File_set_view (Syntax)

- Describes that part of the file accessed by a single MPI process.
- Arguments to MPI_File_set_view:
  - **MPI_File file**
  - **MPI_Offset disp**
  - **MPI_Datatype etype**
  - **MPI_Datatype filetype**
  - **char *datarep**
  - **MPI_Info info**

Collective IO with MPI-IO

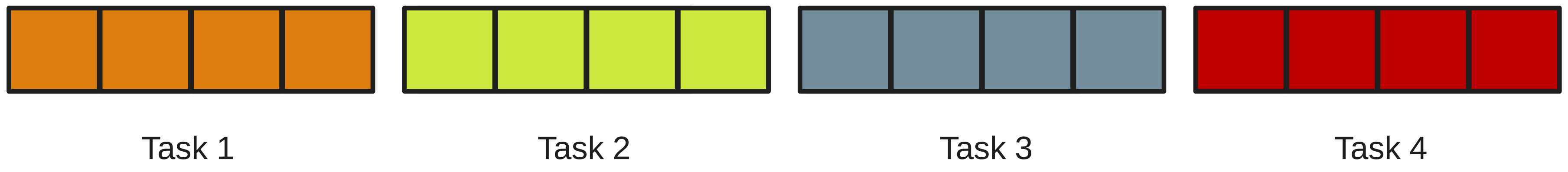
- MPI_File_read**_all**, MPI_File_read_at**_all**, …
- **_all** indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
- Each process specifies only its own access information – the argument list is the same as for the non-collective functions
- MPI-IO library is given a lot of information in this case:
  - Collection of processes reading or writing data
  - Structured description of the regions
- The library has some options for how to use this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

Two techniques : Sieving and Aggregation

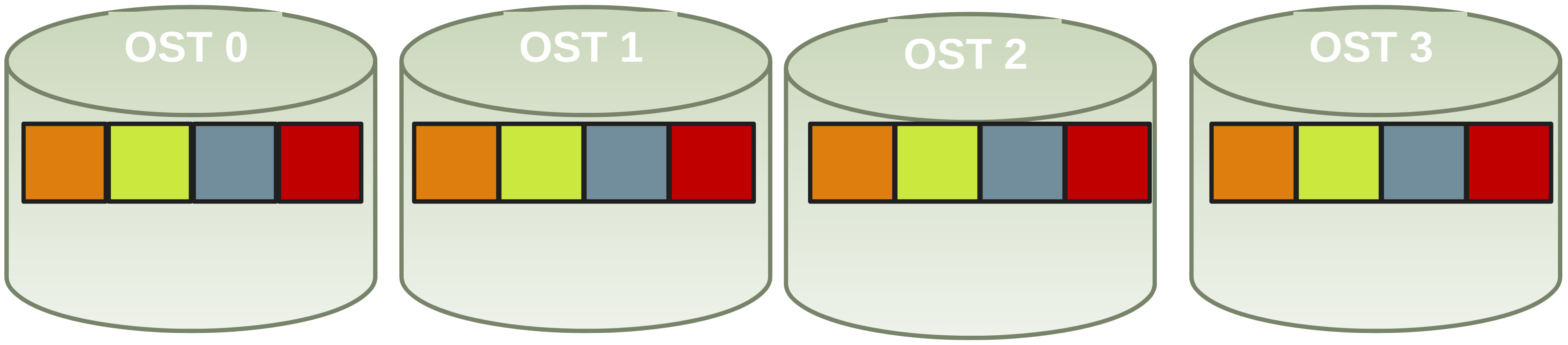
- Data sieving is used to combine lots of small accesses into a single larger one
  - Reducing # of operations important (latency)
  - A system buffer/cache is one example
- Aggregation refers to the concept of moving data through intermediate nodes
  - Different numbers of nodes performing I/O (transparent to the user)
- Both techniques are used by MPI-IO and triggered with HINTS.

Lustre problem : „OST Sharing“

- A file is written by several tasks :

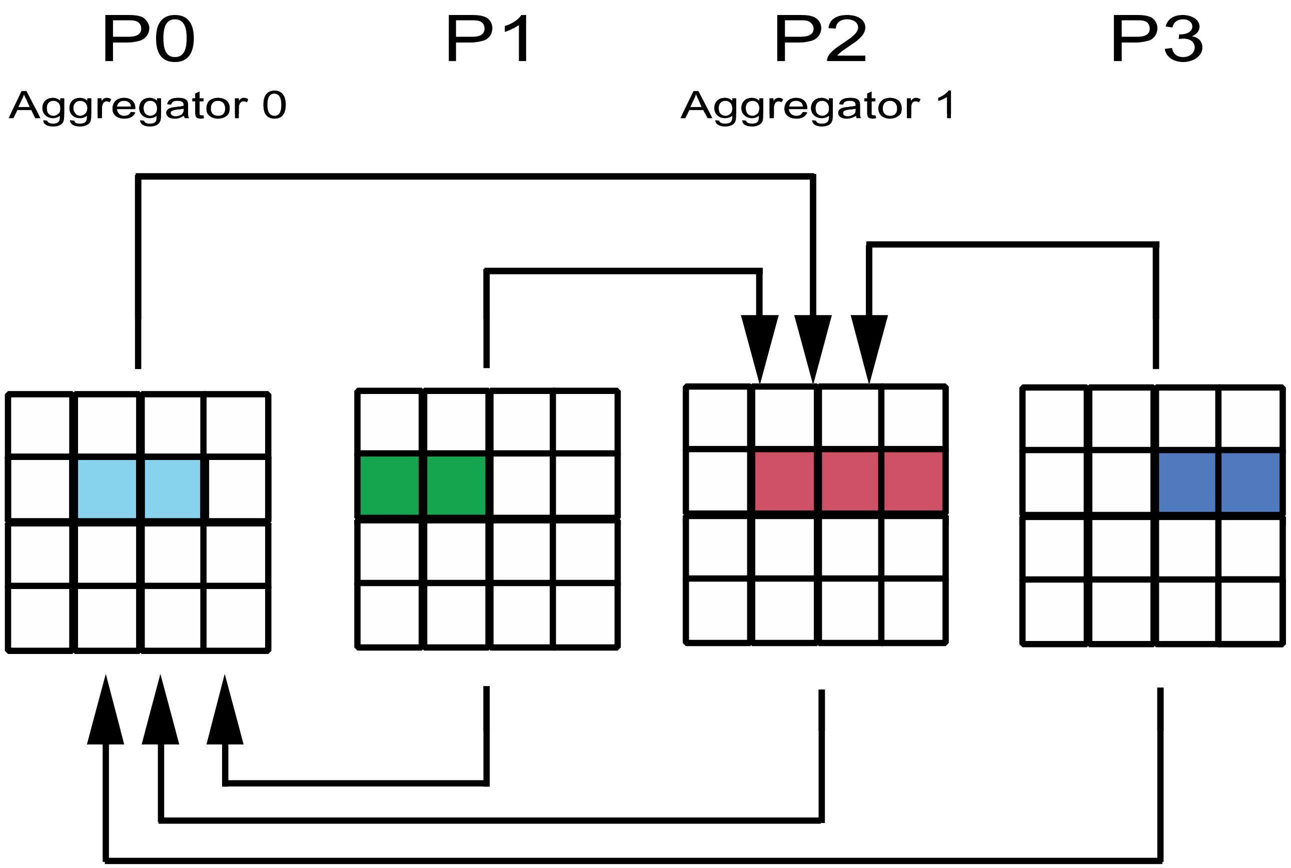


- The file is stored like this (one single stripe per OST for all tasks) :

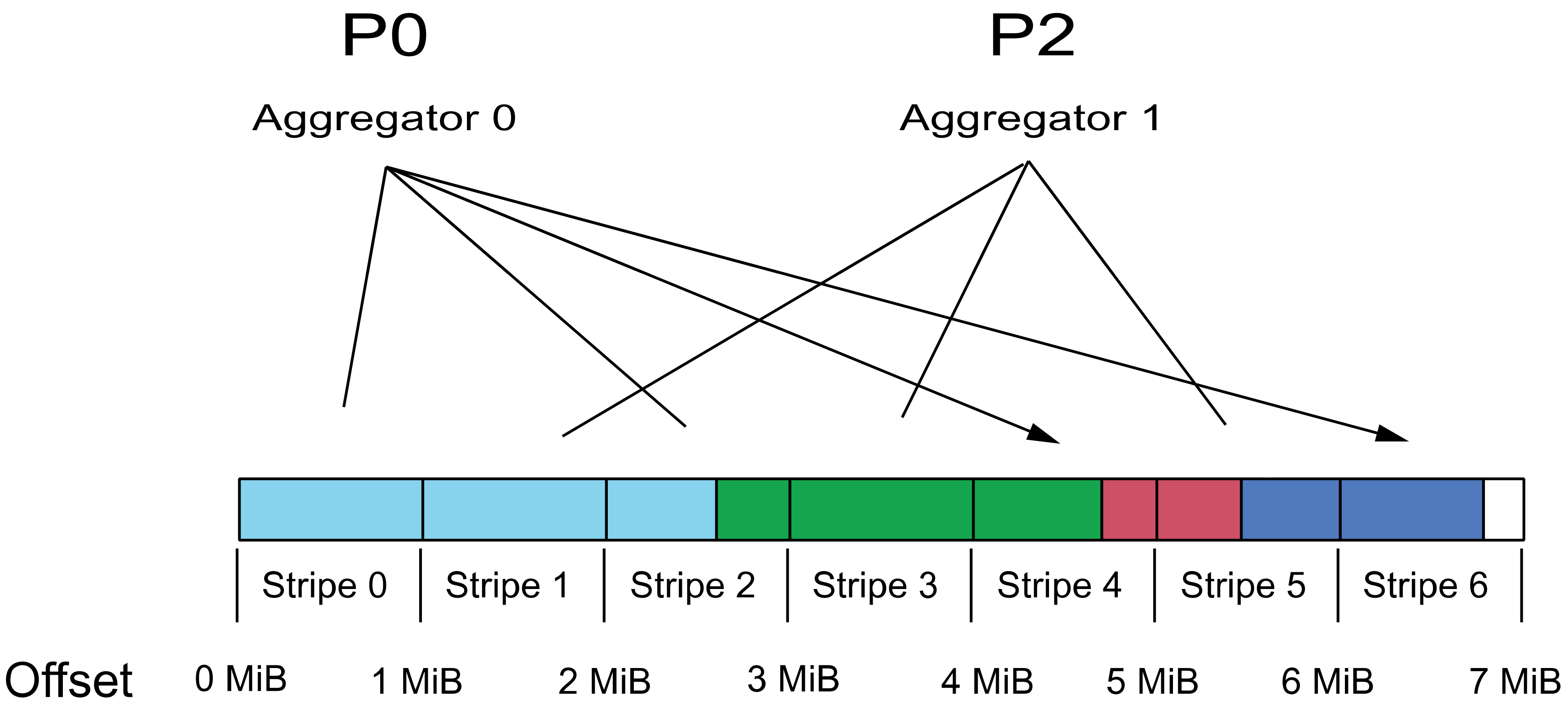


- => Performance Problem (like ‚False Sharing‘ in thread progamming)
- flock mount option needed. only 1 task can write to an OST any time

Collective buffering: aggregating data



collective buffering: writing data



MPI-IO Interaction with Lustre

- Included in the Cray MPT library.
- Environmental variable used to help MPI-IO optimize I/O performance.
  - MPICH_MPIIO_CB_ALIGN Environmental Variable. (Default 2)
  - MPICH_MPIIO_HINTS Environmental Variable
  - Can set striping_factor and striping_unit for files created with MPI-IO.
  - If writes and/or reads utilize collective calls, collective buffering can be utilized (romio_cb_read/write) to approximately stripe align I/O within Lustre.
- HDF5 and NETCDF are both implemented on top of MPI-IO and thus also uses the MPI-IO env. Variables.

MPICH_MPIIO_CB_ALIGN: collective buffering

- If set to 2, an algorithm is used to divide the I/O workload into Lustre stripe-sized pieces and assigns them to collective buffering nodes (aggregators), so that each aggregator always accesses the same set of stripes and no other aggregator accesses those stripes. If the overhead associated with dividing the I/O workload can in some cases exceed the time otherwise saved by using this method.
- If set to 1, an algorithm is used that takes into account physical I/O boundaries and the size of I/O requests in order to determine how to divide the I/O workload when collective buffering is enabled. However, unlike mode 2, there is no fixed association between file stripe and aggregator from one call to the next.
- If set to zero or defined but not assigned a value, an algorithm is used to divide the I/O workload equally amongst all aggregators without regard to physical I/O boundaries or Lustre stripes.

MPI-IO Hints (part 1)

- **MPICH_MPIIO_HINTS_DISPLAY** – Rank 0 displays the name and values of the MPI-IO hints
- **MPICH_MPIO_HINTS** – Sets the MPI-IO hints for files opened with the MPI_File_Open routine
  - Overrides any values set in the application by the MPI_Info_set routine
  - Following hints supported:

direct_io	cb_nodes	romio_ds_write
romio_cb_read	cb_config_list	ind_rd_buffer_size
romio_cb_write	romio_no_indep_rw	Ind_wr_buffer_size
cb_buffer_size	romio_ds_read	striping_factor
		striping_unit

THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

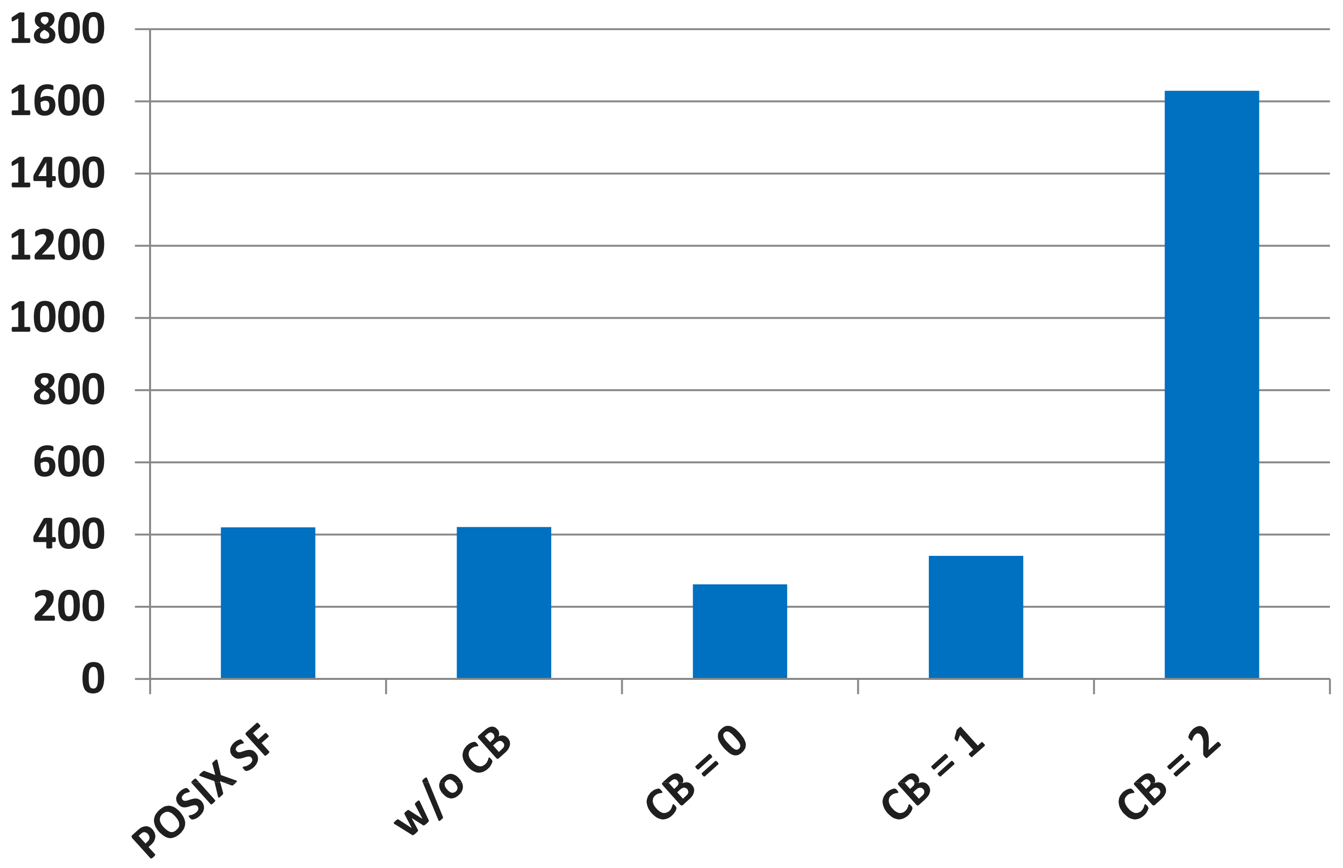
HP2C

Env. Variable MPICH_MPIO_HINTS (part 2)

- If set, override the default value of one or more MPI I/O hints. This also overrides any values that were set by using calls to MPI_Info_set in the application code. The new values apply to the file the next time it is opened using a MPI_File_open() call.
- After the MPI_File_open() call, subsequent MPI_Info_set calls can be used to pass new MPI I/O hints that take precedence over some of the environment variable values. Other MPI I/O hints such as striping factor, striping_unit, cb_nodes, and cb_config_list cannot be changed after the MPI_File_open() call, as these are evaluated and applied only during the file open process.
- The syntax for this environment variable is a comma-separated list of specifications. Each individual specification is a pathname_pattern followed by a colon-separated list of one or more key=value pairs. In each key=value pair, the key is the MPI-IO hint name, and the value is its value as it would be coded for an MPI_Info_set library call.
- Example:
  MPICH_MPIIO_HINTS=file1:direct_io=true,file2:romio_ds_write=disable,/scratch/user/me/dump.*:romio_cb_write=enable:cb_nodes=8

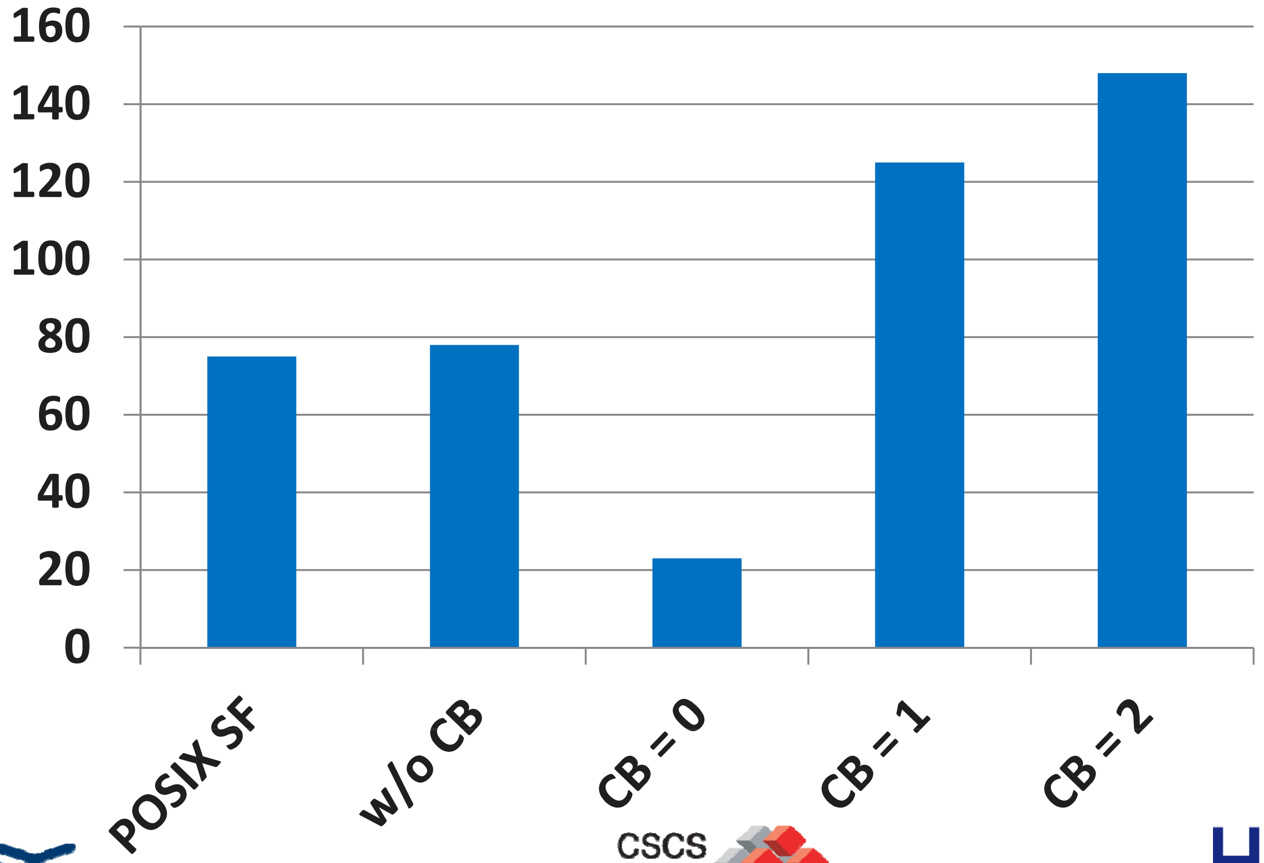
IOR benchmark 1,000,000 bytes

MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 1M bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file



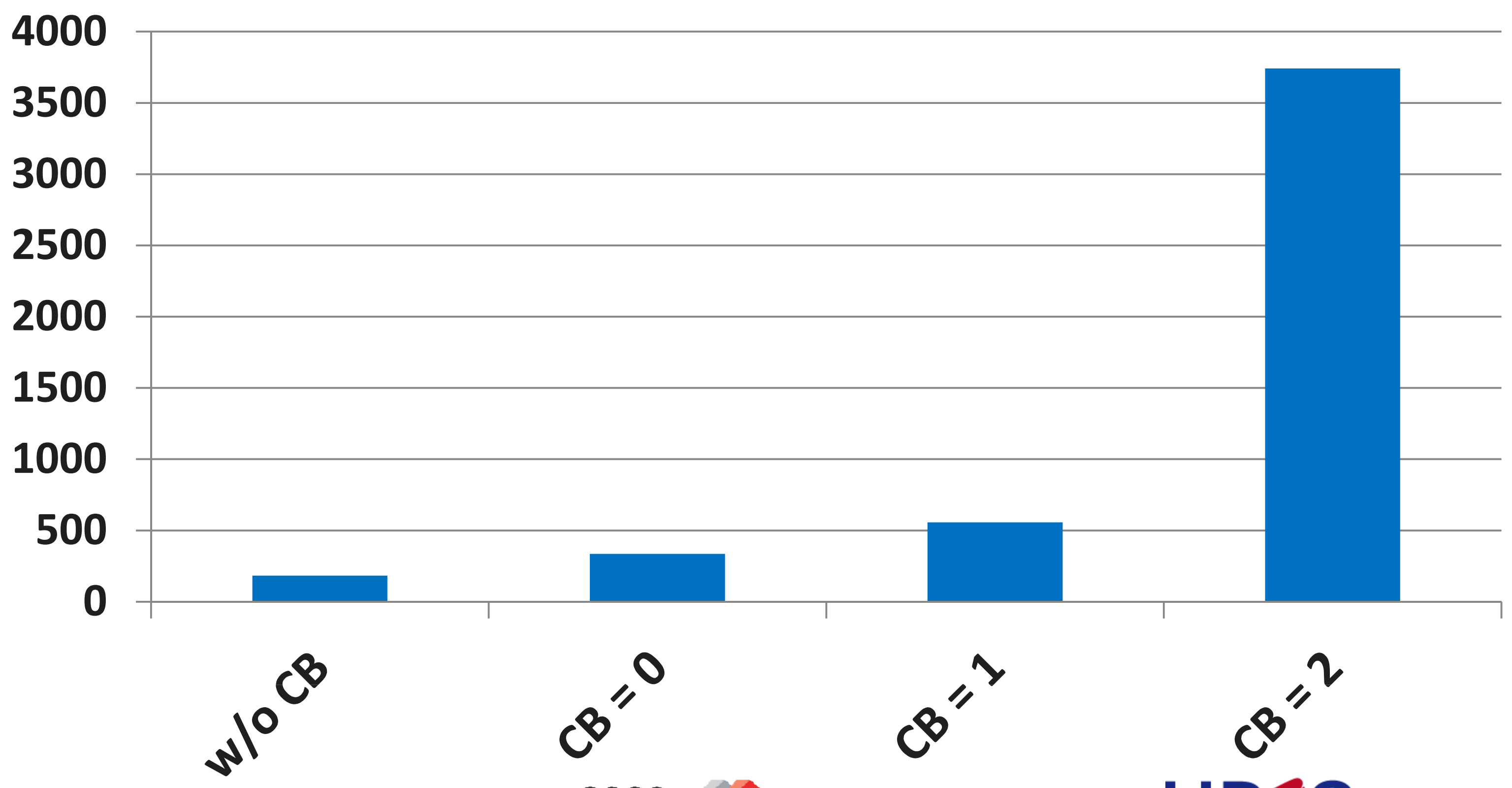
IOR benchmark 10,000 bytes

MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 10K bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file



HYCOM MPI-2 I/O

On 5107 PEs, and by application design, a subset of the PEs(88), do the writes. With collective buffering, this is further reduced to 22 aggregators (cb_nodes) writing to 22 stripes. Tested on an XT5 with 5107 PEs, 8 cores/node



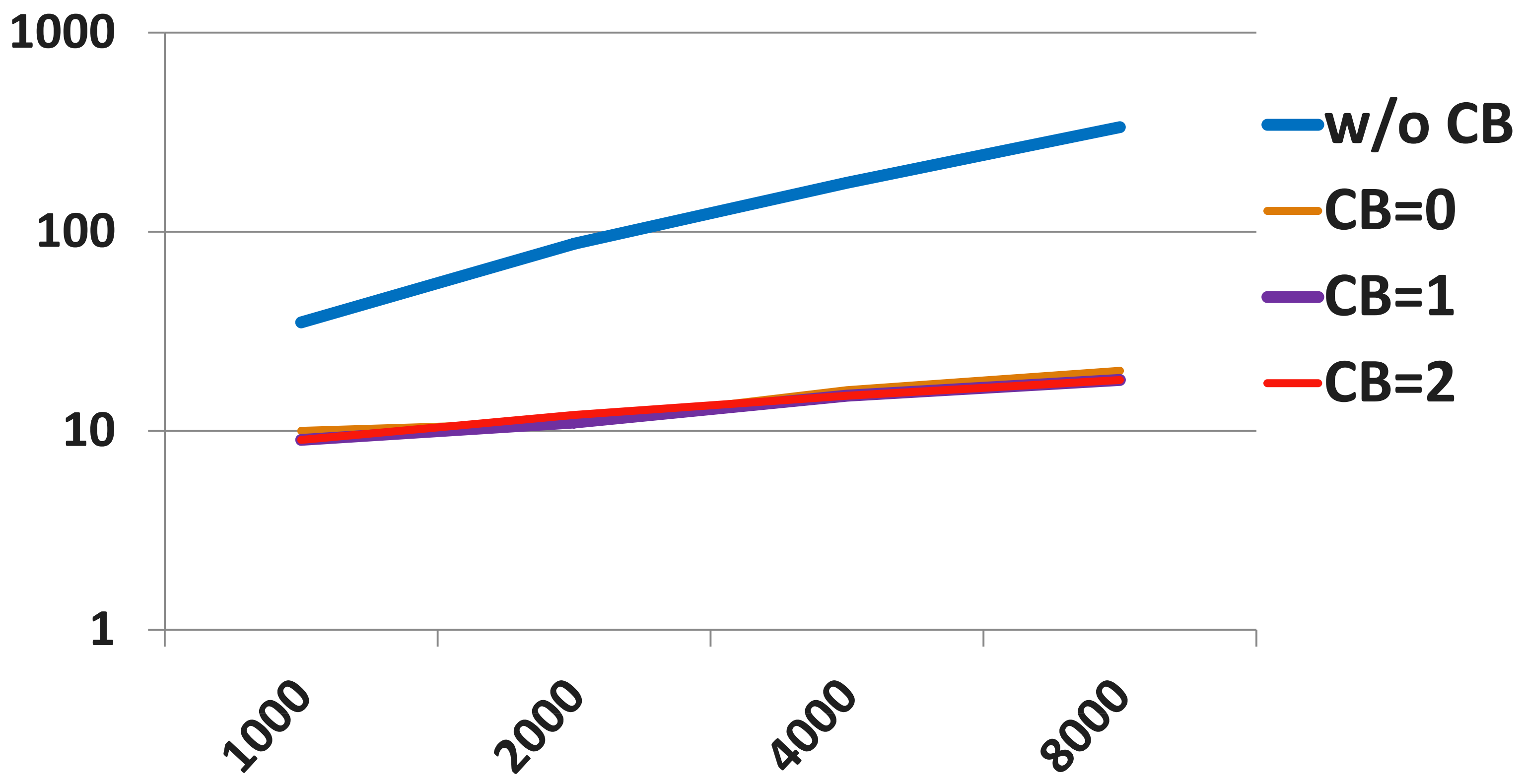
CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

HDF5 format dump file from all PEs

Total file size 6.4 GiB. Mesh of 64M bytes 32M elements, with work divided amongst all PEs. Original problem was very poor scaling. For example, without collective buffering, 8000 PEs take over 5 minutes to dump. Note that disabling data sieving was necessary. Tested on an XT5, 8 stripes, 8 cb_nodes



MPI-IO Example

Storing a distributed Domain into a single File

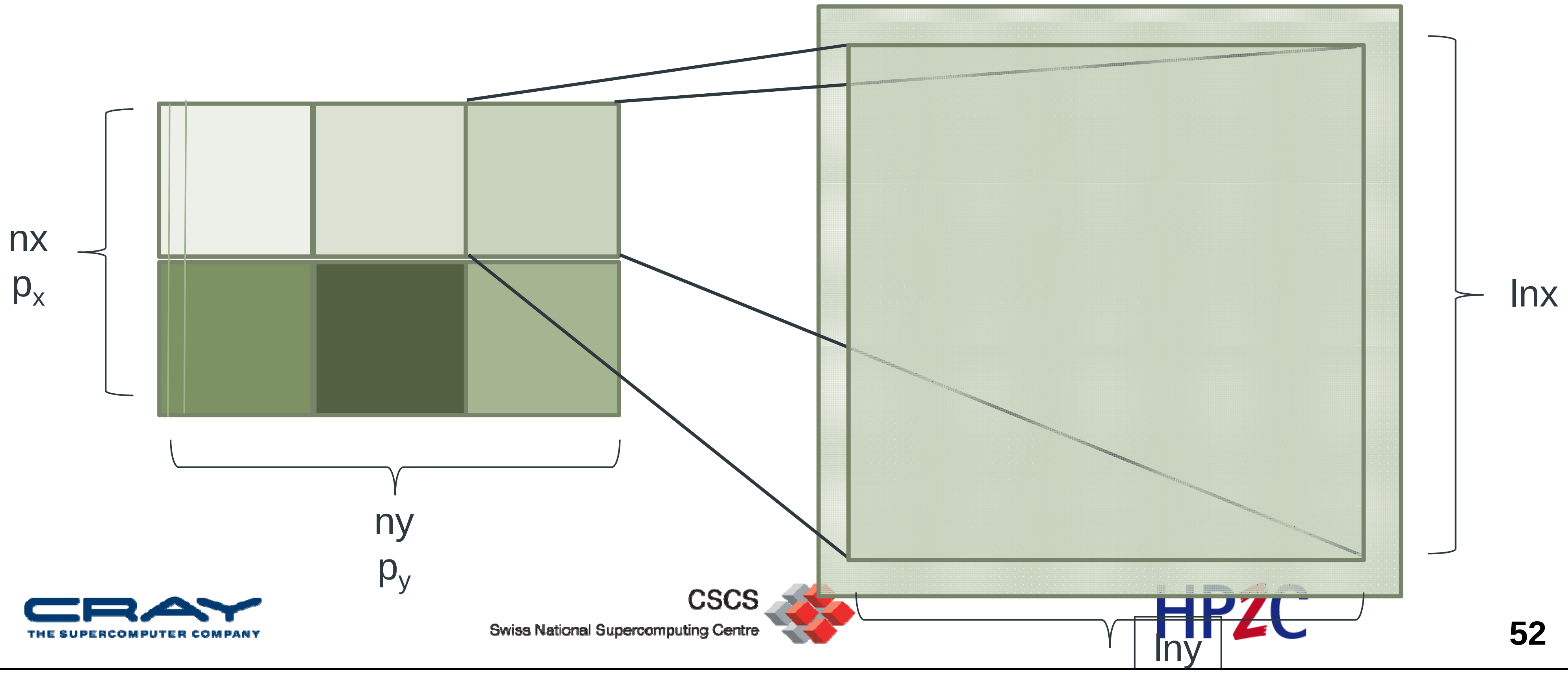
CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Problem we want to solve

- We have 2 dim domain on a 2 dimensional processor grid
- Each local subdomain has a halo (ghost cells).
- The data (without halo) is going to be stored in a single file, which can be re-read by any processor count
- Here an example with 2x3 procesor grid :



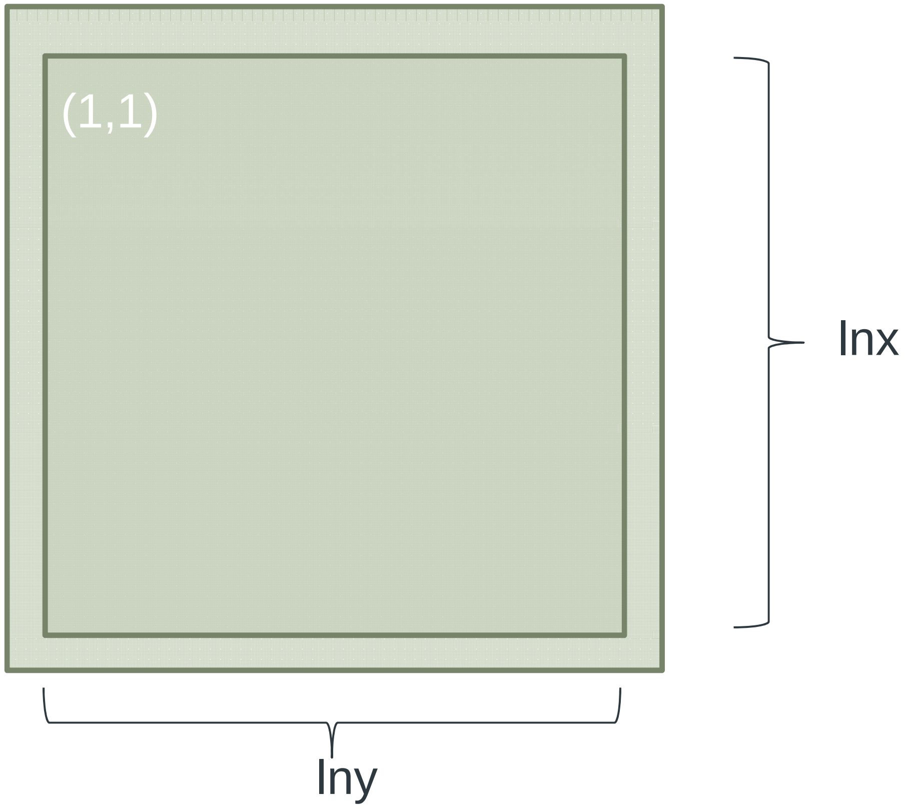
Approach for writing the file

- First step is to create the MPI 2 dimensional processor grid
- Second step is to describe the local data layout using a MPI datatype
- Then we create a „global MPI datatype“ describing how the data should be stored
- Finaly we do the I/O

Basic MPI setup

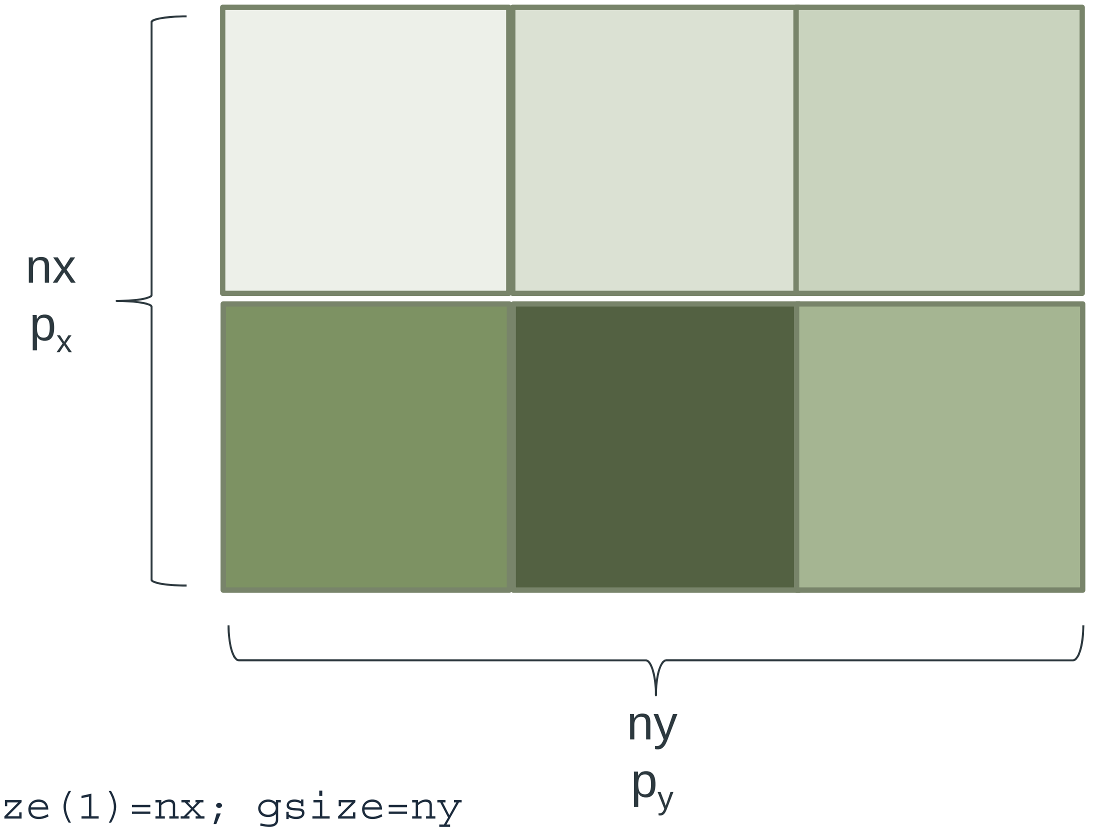
```
nx=512; ny=512 ! Global Domain Size
call MPI_Init(mpierr)
call MPI_Comm_size(MPI_COMM_WORLD, mysize, mpierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, mpierr)

dom_size(1)=2;  dom_size(2)=mysize/dom_size(1)
lnx=nx/dom_size(1) ;  lny=ny/dom_size(2) ! Local Domain size
periods=.false. ; reorder=.false.
call MPI_Cart_create(MPI_COMM_WORLD, dim, dom_size, periods, reorder,
  comm_cart, mpierr)
call MPI_Cart_coords(comm_cart, myrank, dim, my_coords, mpierr)

halo=1
allocate (domain(0:lnx+halo, 0:lny+halo))
```

Creating the local data type



```
gsize(1)=lnx+2; gsize(2)=lny+2
lsize(1)=lnx; lsize(2)=lny
start(1)=1; start(2)=1
call MPI_Type_create_subarray(dim, gsize, lsize, start,
 MPI ORDER FORTRAN, MPI INTEGER, type local, mpierr)
call MPI_Type_commit(type_local, mpierr)
```

And now the global datatype



```
gsize(1)=nx; gsize=ny
lsize(1)=lnx; lsize(2)=lny
start(1)=lnx*my_coords(1); start(2)=lny*my_coords(2)
call MPI_Type_create_subarray(dim, gsize, lsize, start,
 MPI ORDER FORTRAN, MPI INTEGER, type domain, mpierr)
call MPI_Type_commit(type_domain, mpierr)
```

Now we have all together

```
call MPI_Info_create(fileinfo, mpierr)
call MPI_File_delete('FILE', MPI_INFO_NULL, mpierr)
call MPI_File_open(MPI_COMM_WORLD, 'FILE',
 IOR(MPI_MODE_RDWR,MPI_MODE_CREATE), fileinfo, fh, mpierr)

disp=0 ! Note : INTEGER(kind=MPI OFFSET KIND) :: disp
call MPI_File_set_view(fh, disp, MPI_INTEGER, type_domain, 'native',
 fileinfo, mpierr)
call MPI_File_write_all(fh, domain, 1, type_local, status, mpierr)
call MPI File close(fh, mpierr)
```

I/O Performance Summary

- Buy sufficient I/O hardware for the machine
  - As your job grows, so does your need for I/O bandwidth
  - You might have to change your I/O implementation when scaling
- Lustre
  - Minimize contention for file system resources.
  - A single process should not access more than 4 OSTs, less might be better
- Performance
  - Performance is limited for single process I/O.
  - Parallel I/O utilizing a file-per-process or a single shared file is limited at large scales.
  - Potential solution is to utilize multiple shared file or a subset of processes which perform I/O.
  - A dedicated I/O Server process (or more) might also help
  - Did not really talk about the MDS

And there is more

- http://docs.cray.com
  - Search for MPI-IO : „Getting started with MPI I/O“, „Optimizing MPI-IO for Applications on CRAY XT Systems“
  - Search for lustre (a lot for admins but not only)
  - Message Passing Toolkit
- Man pages (man mpi, man <mpi_routine>, ...)
- mpich2 standard : http://www.mcs.anl.gov/research/projects/mpich2/