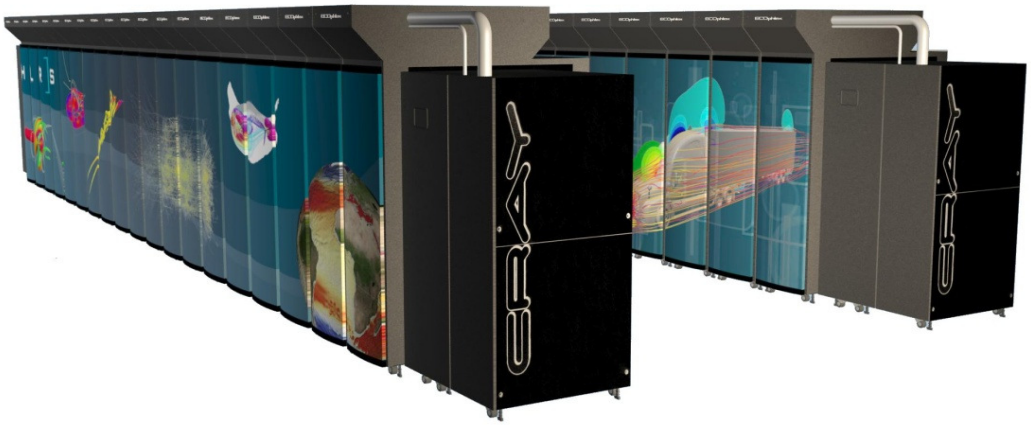


I/O Optimization

Stefan Andersson
stefan@cray.com

CRAY THE SUPERCOMPUTER COMPANY

H L R S

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

Ken Batcher

CRAY THE SUPERCOMPUTER COMPANY

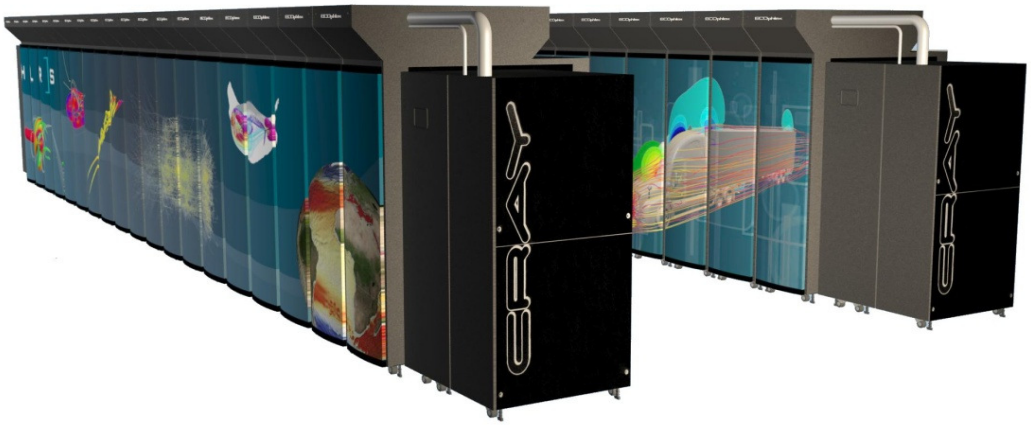
HLRS

Factors which affect I/O : Know your system and code

- I/O is simply data migration.
  - Memory ⟷ Disk
- I/O is a very expensive operation.
  - Interactions with data in memory and on disk.
  - Must get the kernel involved
- How is I/O performed?
  - I/O Pattern
    - Number of processes and files.
    - File access characteristics.
- Where is I/O performed?
  - Characteristics of the computational system.
  - Characteristics of the file system.

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
- IO Interfaces (short)
  - What is there
- Using MPI-IO
  - Performance and a few short code examples
- A non trivial MPI-IO example

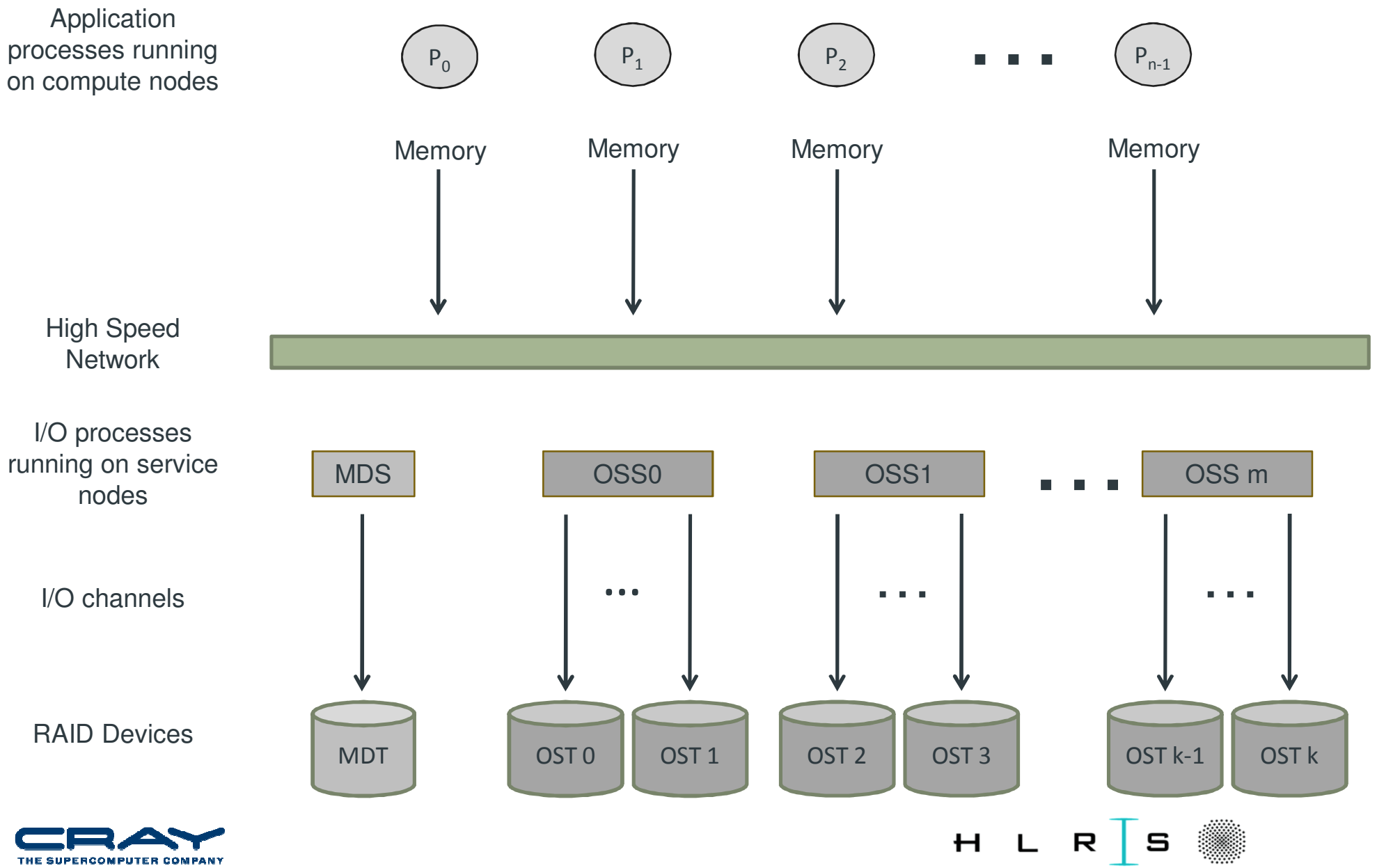


Basics about Lustre

CRAY THE SUPERCOMPUTER COMPANY

H L R S

Basic Lustre Overview



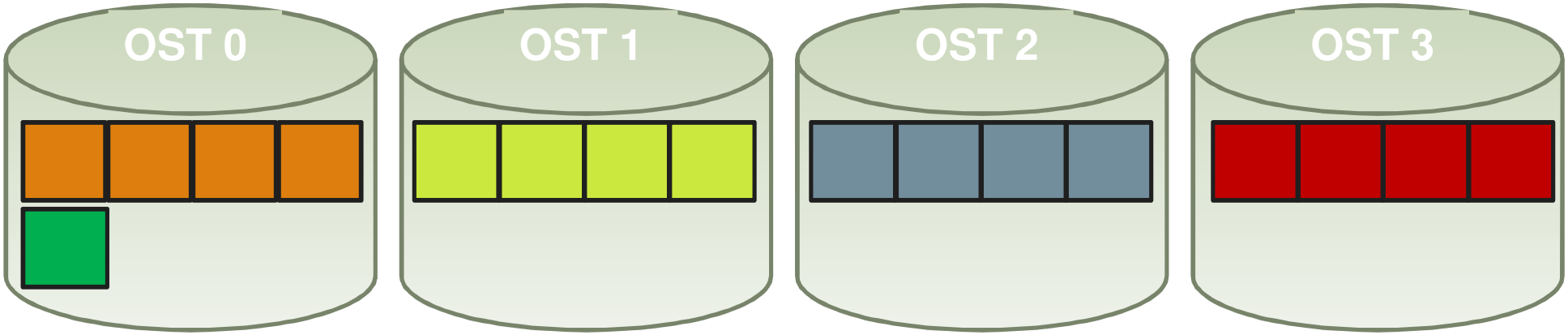
CRAY THE SUPERCOMPUTER COMPANY

HLRS

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.



Lustre Striping

- The user can tell lustre how to stripe a file
- 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 absulte maximum of 160
- The **„Stripe Index“** is the starting OST of the file
- You control the striping by the **„lfs“** command (see next slide)
- The application does not directly reference OSTs or physical I/O blocks

Setting the stripe values

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

```
stefan@seal3:~> lfs 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 <filename|dirname> <stripe_size> <stripe_index> <stripe_count>
 or
setstripe <filename|dirname> [--size|-s stripe_size]
                             [--index|-i stripe_index]
                             [--count|-c stripe_count]
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)
stripe_index: OST index of first stripe (-1 filesystem default)
stripe_count: Number of OSTs to stripe over (0 default, -1 all)
```

- 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 how the striping values are set

- When creating a directory, it get the default lustre settings
  - You can change this anytime. A change will not effect existing files in the directory
- A file/directory will inherit the lustre setting of the directory it is created in
- 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 exisiting file you have to copy it :

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

Getting the stripe values

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

```
 stefan@seal3:> touch delme
stefan@seal3:> lfs getstripe delme
OBDS:
0: ost0_UUID ACTIVE
1: ost1_UUID ACTIVE
2: ost2_UUID ACTIVE
3: ost3_UUID ACTIVE
4: ost4_UUID ACTIVE
5: ost5_UUID ACTIVE
6: ost6_UUID ACTIVE
7: ost7_UUID ACTIVE
delme
        obdidx           objid           objid             group
             2        56309996      0x35b38ec                 0
             1        56662062      0x360982e                 0
stefan@seal3:>
```

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:~>
```

And lfs can more. Check the build-in help

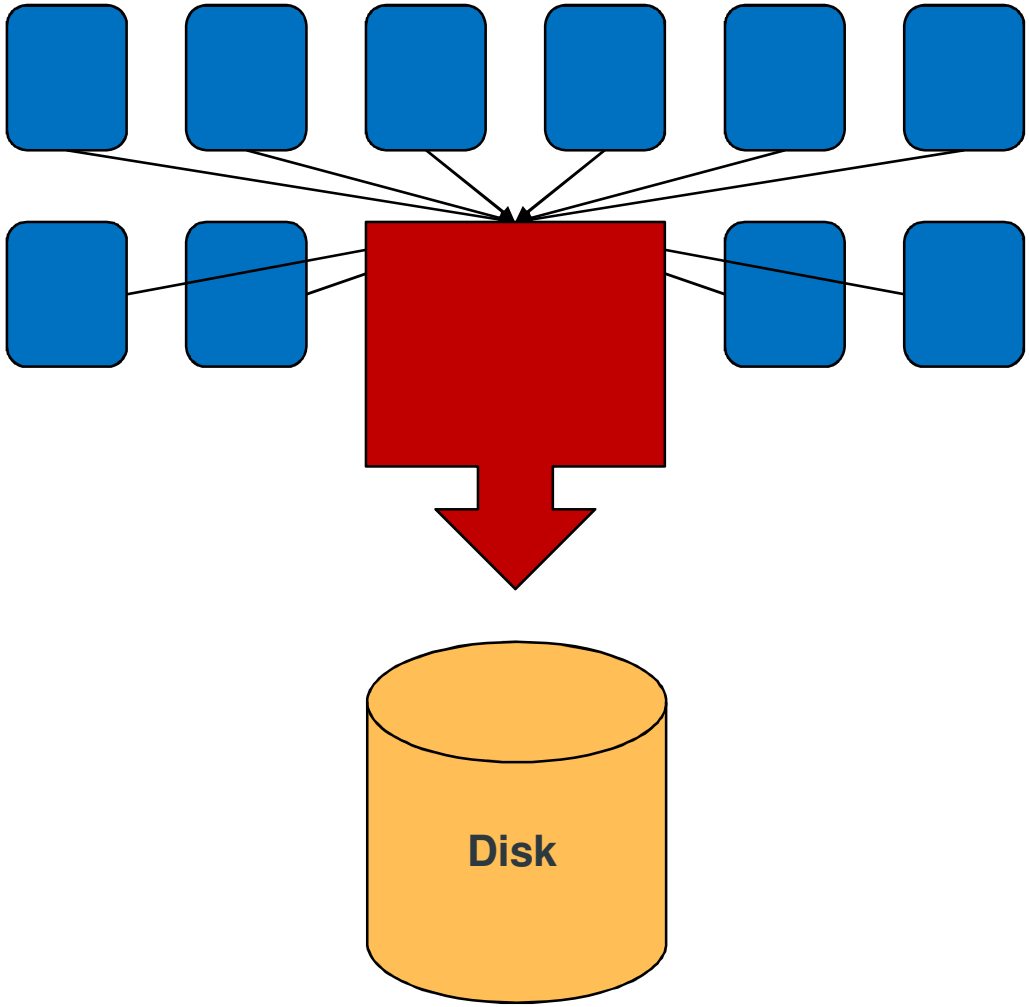
```
stefan@seal1:~> lfs
   lfs > help
   Available commands are:
            setstripe
            getstripe
            find
            check
            catinfo
            join
            osts
            df
            quotachown
            quotacheck
            quotaon
            quotaoff
            setquota
            quota
            quotainv
            help
            exit
            quit
   For more help type: help command-name
   lfs >
```

I/O Strategies

How can parallel I/O be done

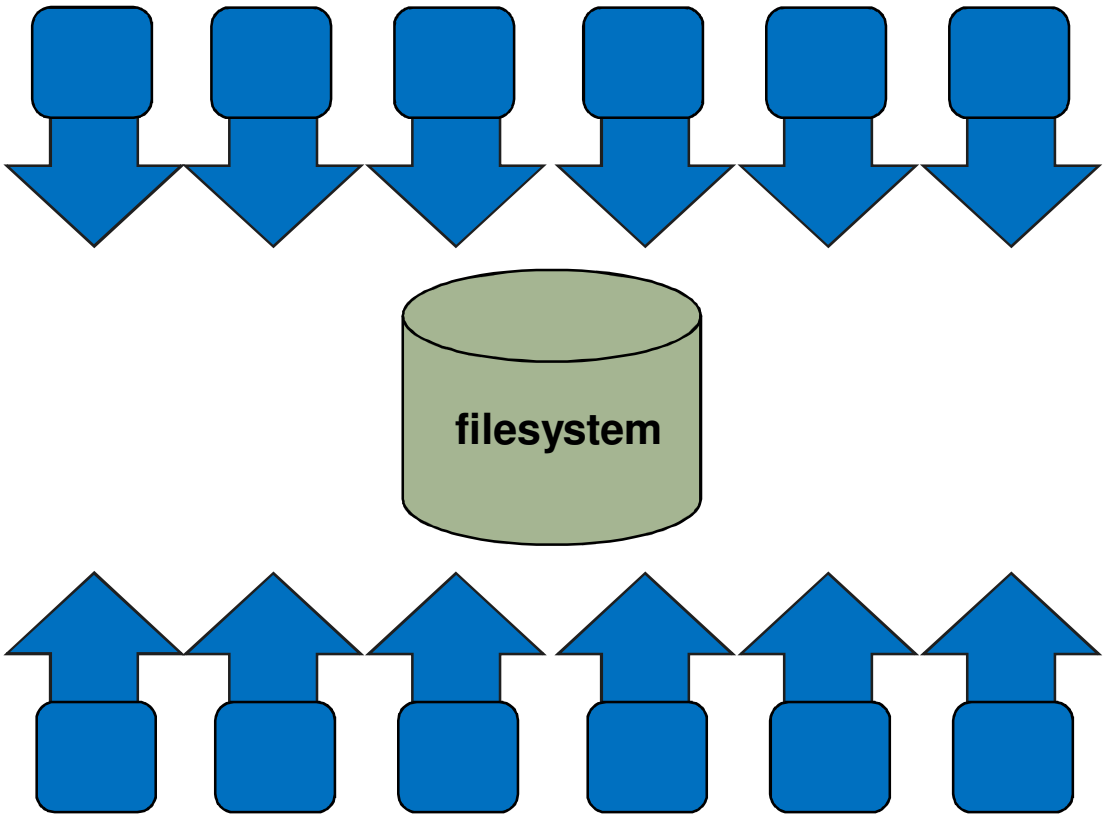
Spokesperson, basically serial I/O

- 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)



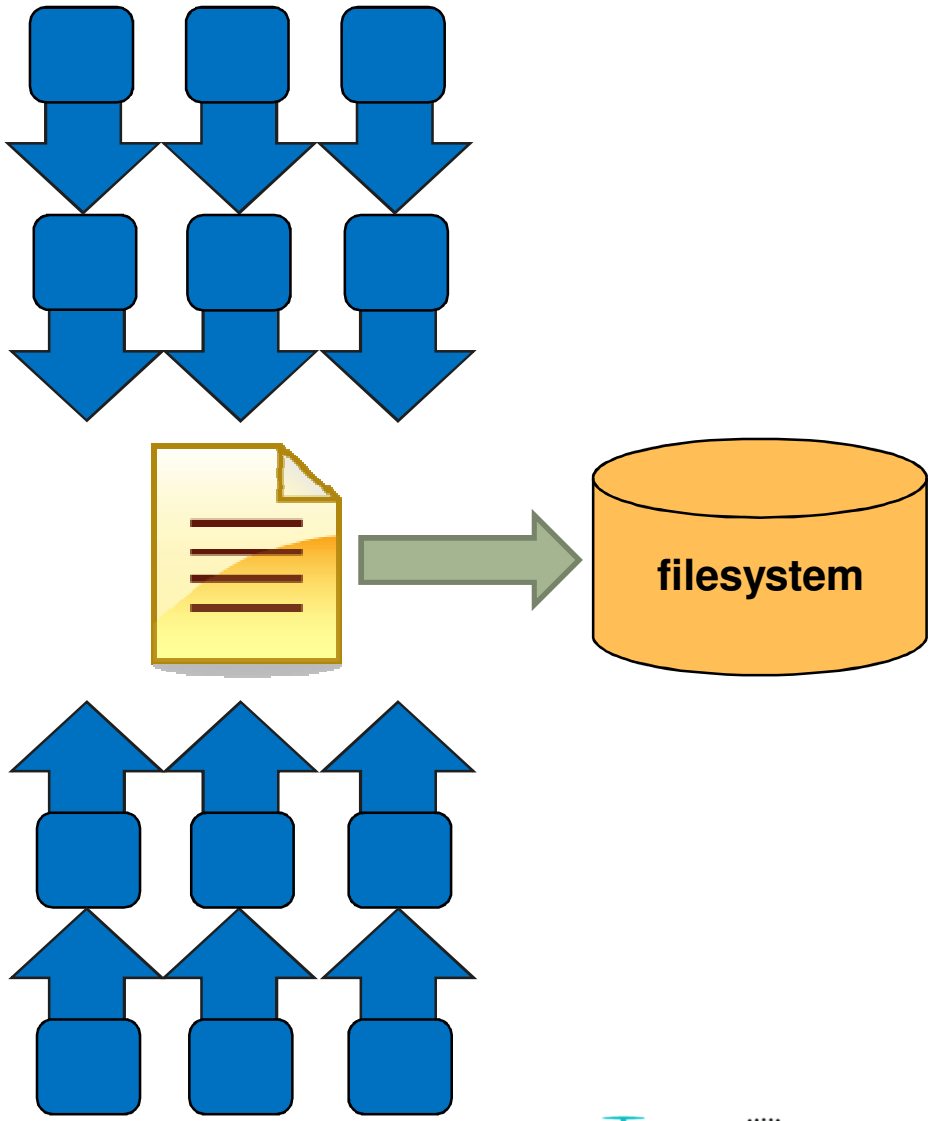
Single File per process

- 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.



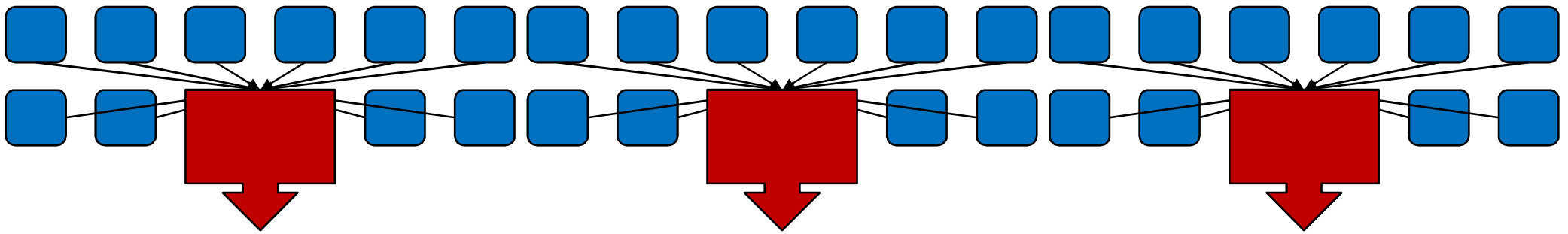
Shared File

- 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



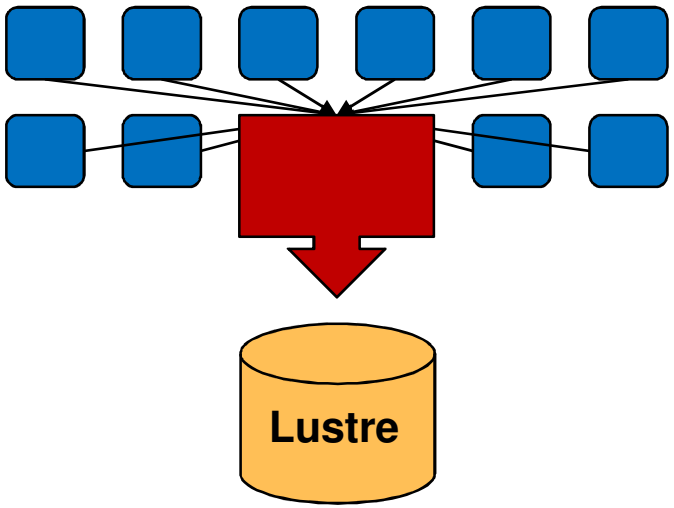
A little bit of all, using a subset of processes

- 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

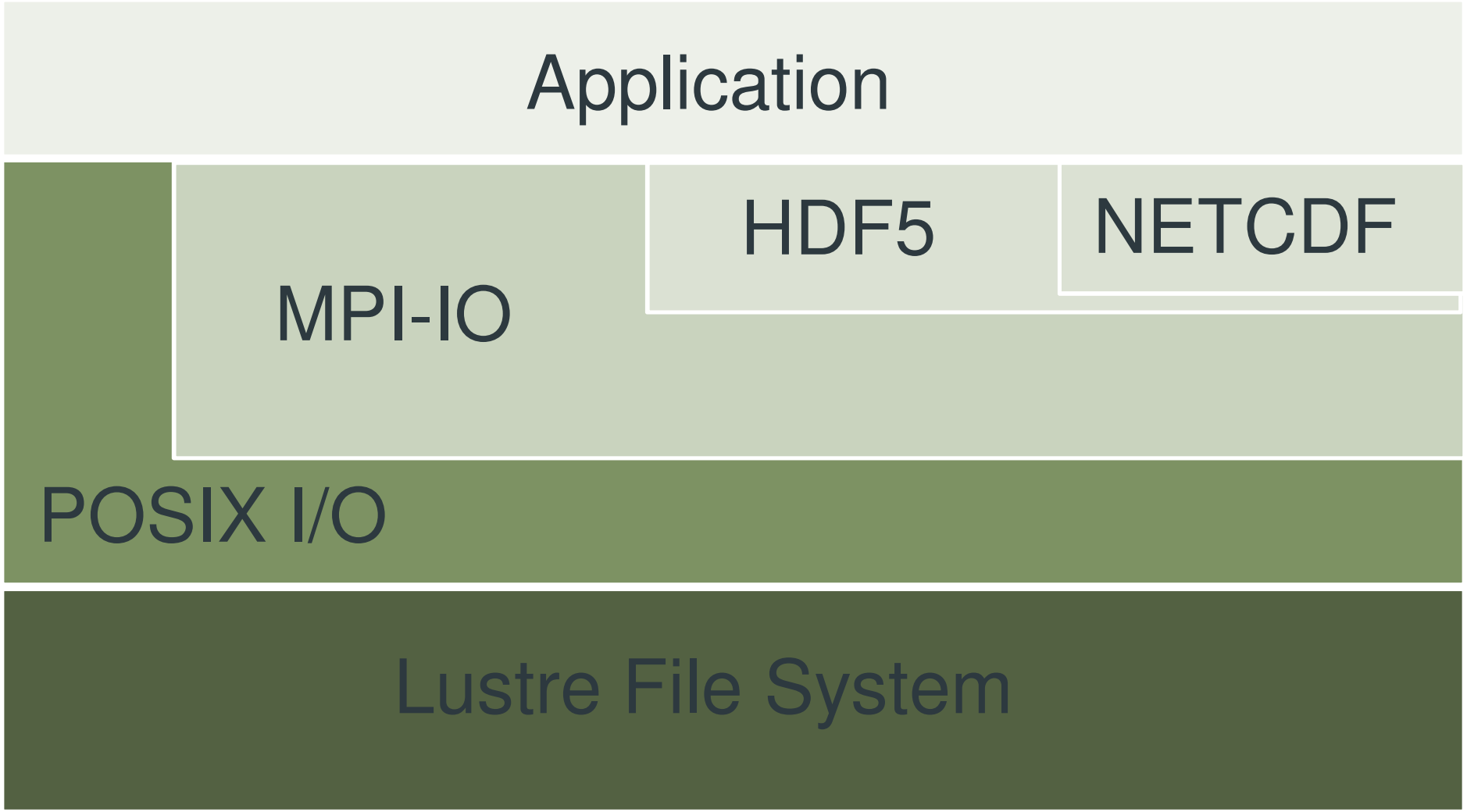
- Standard Output and Error streams are effectively serial I/O.
- All STDIN, STDOUT, and STDERR I/O serialize through aprun
- Disable debugging messages when running in production mode.
  - “Hello, I’m task 32000!”
  - “Task 64000, made it through loop.”
  - …



IO Interfaces

How do I program IO

CRAY IO Software stack



IO Interfaces : POSIX I/O

- Lowest level of I/O programming
- File is a sequence of bytes
- Fortran,C and C++ I/O calls are converted to POSIX I/O
- It is not parallel I/O, but it is possible to use in parallel
  - You have to coordinate any parallel access, but it is complicated (closing a file in parallel, when to write, what about buffers, flushing, ...)
- Low overhead and potentially fast
- Suported standard as part of the programmin languages

IO Interfaces : MPI-IO

- MPI-IO can be done in 2 basic ways :
- Independent MPI-IO
  - For independent I/O each MPI task is handling the I/O independently using non collective calls like MPI_File_write() and MPI_File_read().
    Think MPI_Send() and MPI_Recv() with a filesystem as partner
  - Similar to POSIX I/O, but supports derived datatypes and thus noncontiguous data and nonuniform strides and can take advantaeges of MPI_Hints
- Collective MPI-IO
  - When doing collective I/O all MPI tasks participating in I/O has to call the same routines. Basic routines are MPI_File_write_all() and MPI_File_read_all()
  - This allows the MPI library to do IO optimization

IO Interfaces : HDF5 and NETCDF (not covered in this presentation)

- HDF5 is platform-independent I/O that simpliefies the modeling, viewing and analysis of complex data objects. It provides a higher level of data abstractions then MPI-IO
  See http://www.hdfgroup.org
- NETCDF-4 is a platform-independent I/O interface that allows you to create, access and share array-oriented data. NetCDF-4 provides a higher level of data abstraction then MPI-IO.
  See man netcdf(3) and
  http://www.unidata.ucar.edu/software/netcdf/docs

I/O Optimizations

‚outside‘ and ‚inside‘ your application

CRAY THE SUPERCOMPUTER COMPANY

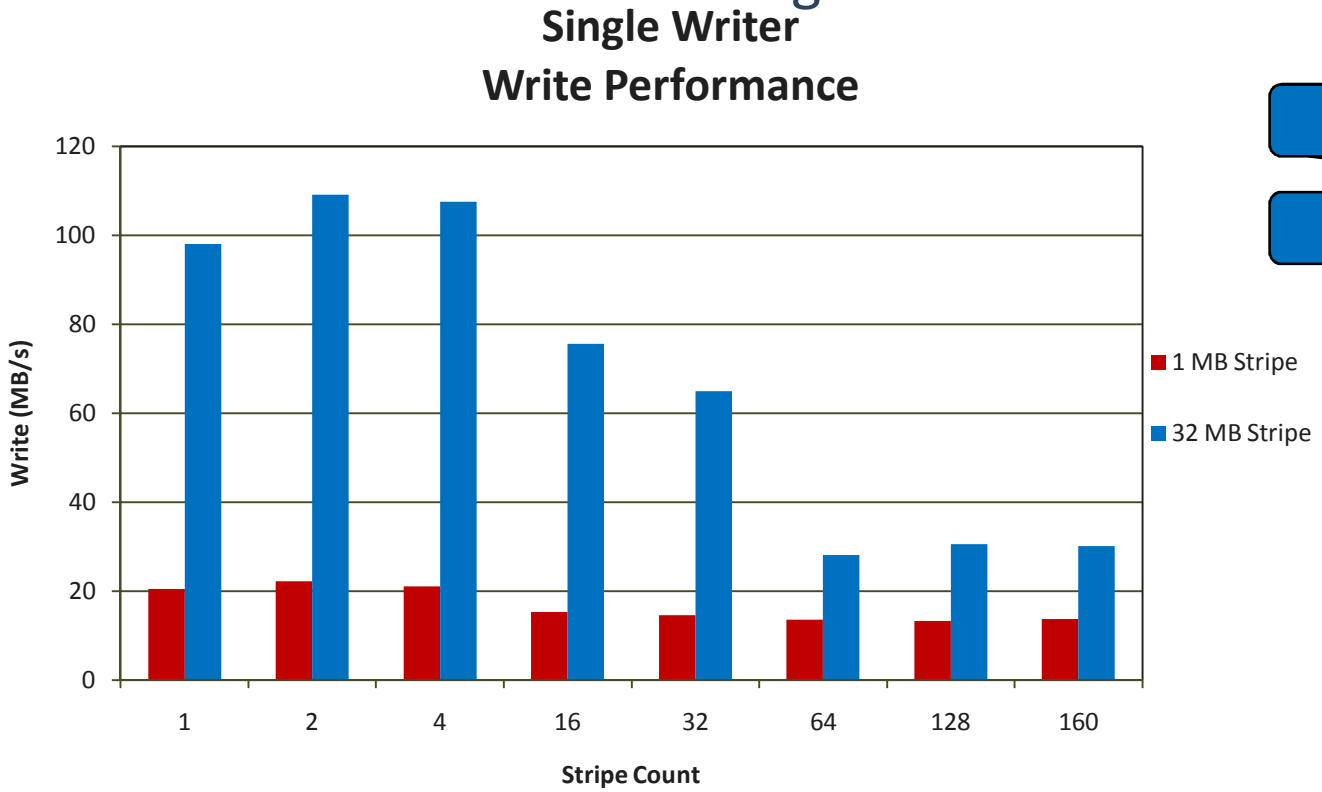
HLRS

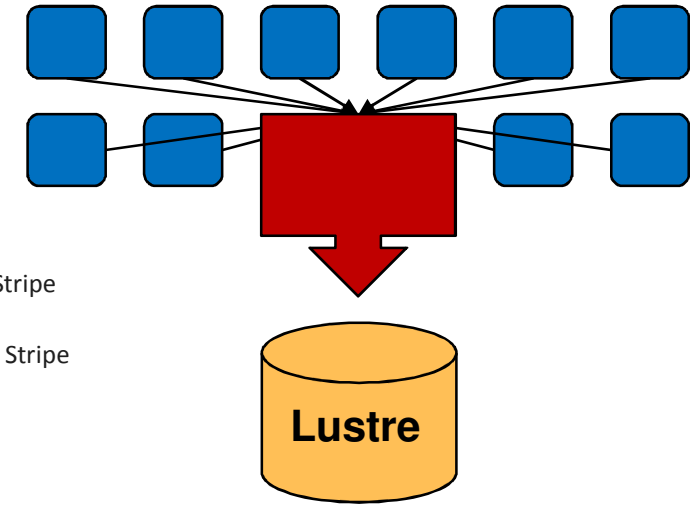
First step : Select best striping values

- 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 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

Case Study 1 : Spokesman

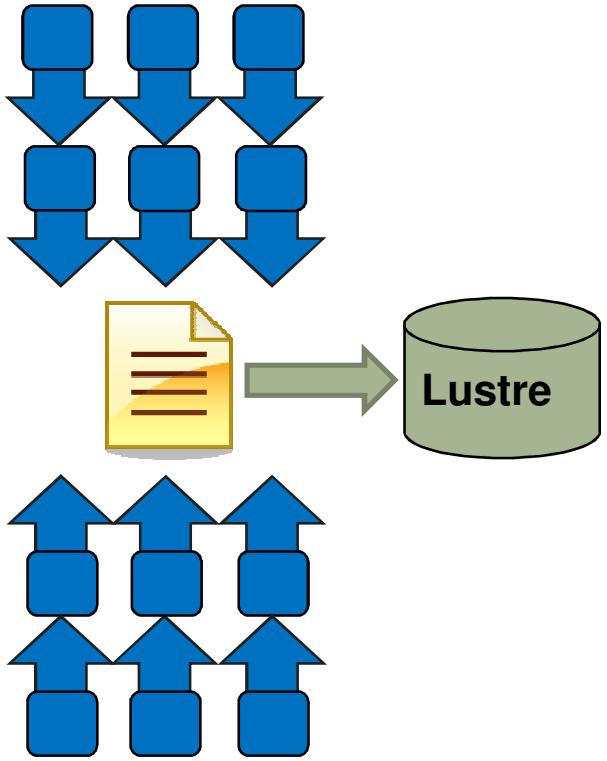
- 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance
  - Note : XE6 numbers might be better





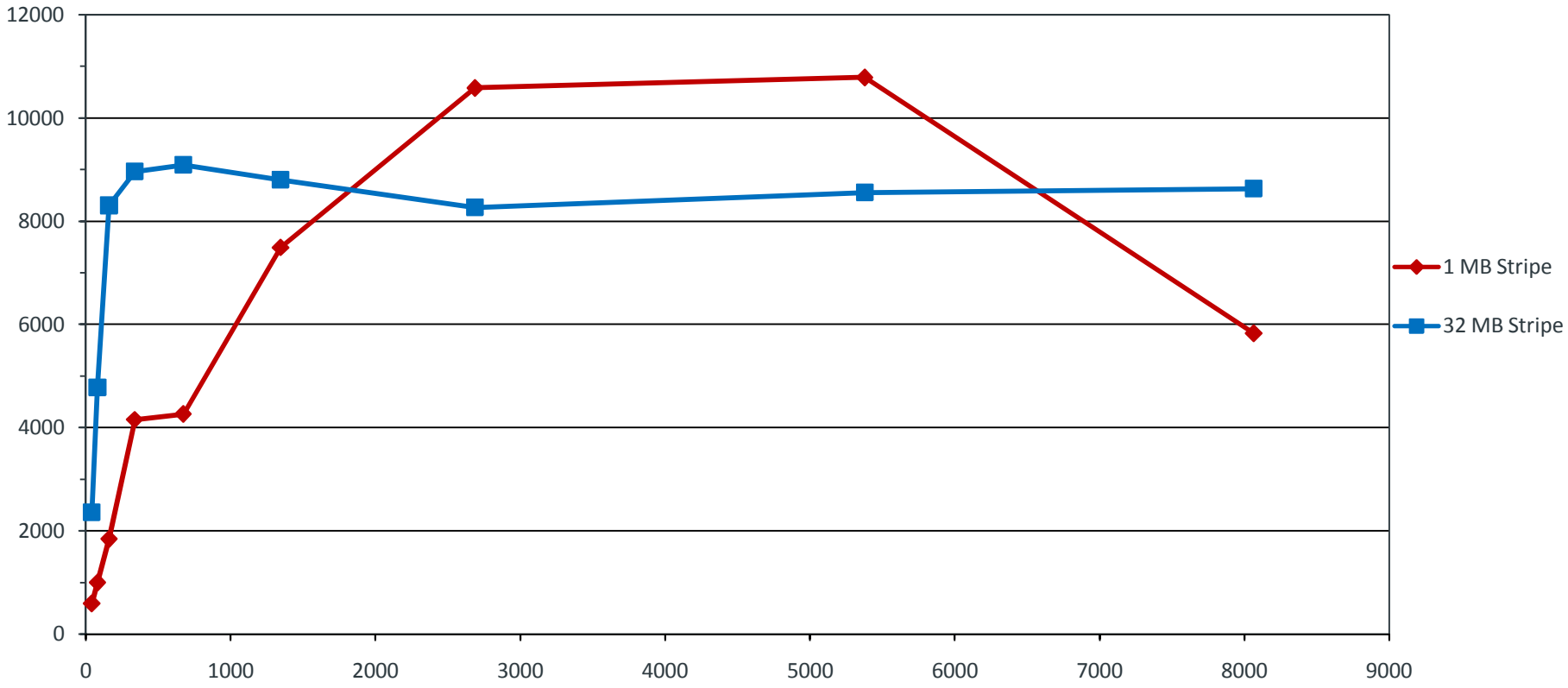
Case Study 2 : Parallel I/O into a single file

- 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



CRAY THE SUPERCOMPUTER COMPANY

HLRS

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 I/O with MPI-IO

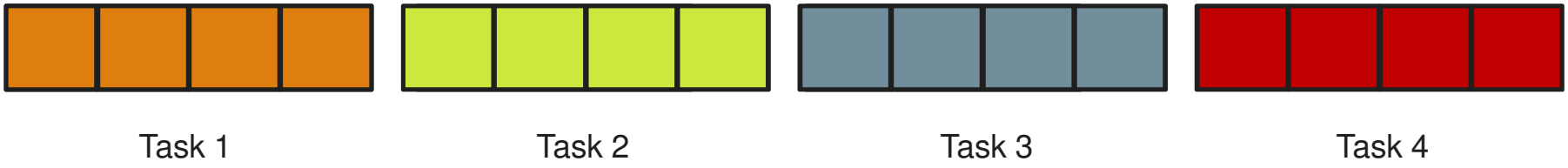
- MPI_File_read**_all**, MPI_File_read_at**_all**, …
- **_all** indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
- Each process specifies only its own access information – the argument list is the same as for the non-collective functions
- 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

2 Techniques : Sieving and Aggregation

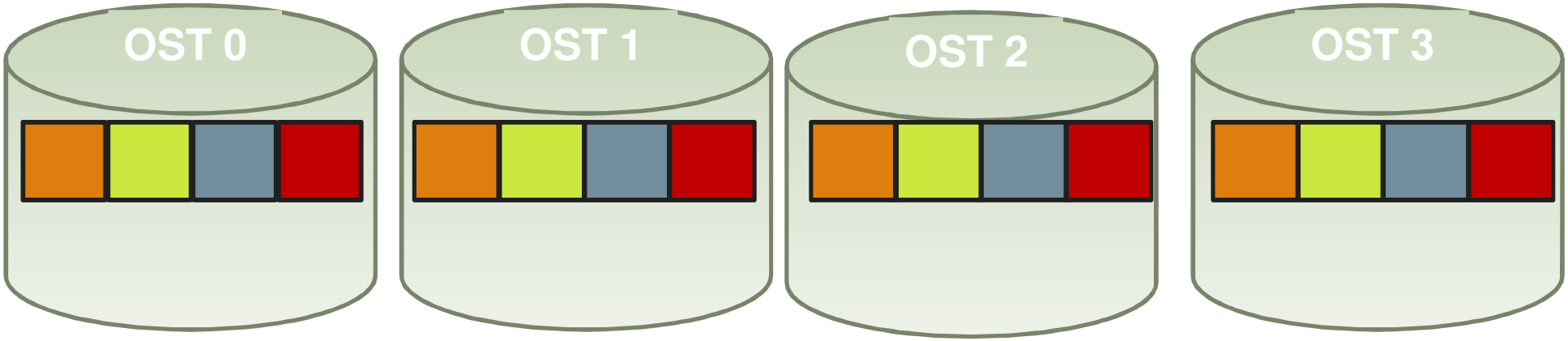
- Data sieving is used to combine lots of small accesses into a single larger one
  - Reducing # of operations important (latency)
  - A system buffer/cache is one example
- Aggregation refers to the concept of moving data through intermediate nodes
  - Different numbers of nodes performing I/O (transparent 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 stripe per OST) :



- => Performance Problem (like ‚False Sharing‘ in thread progamming)

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

- 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:

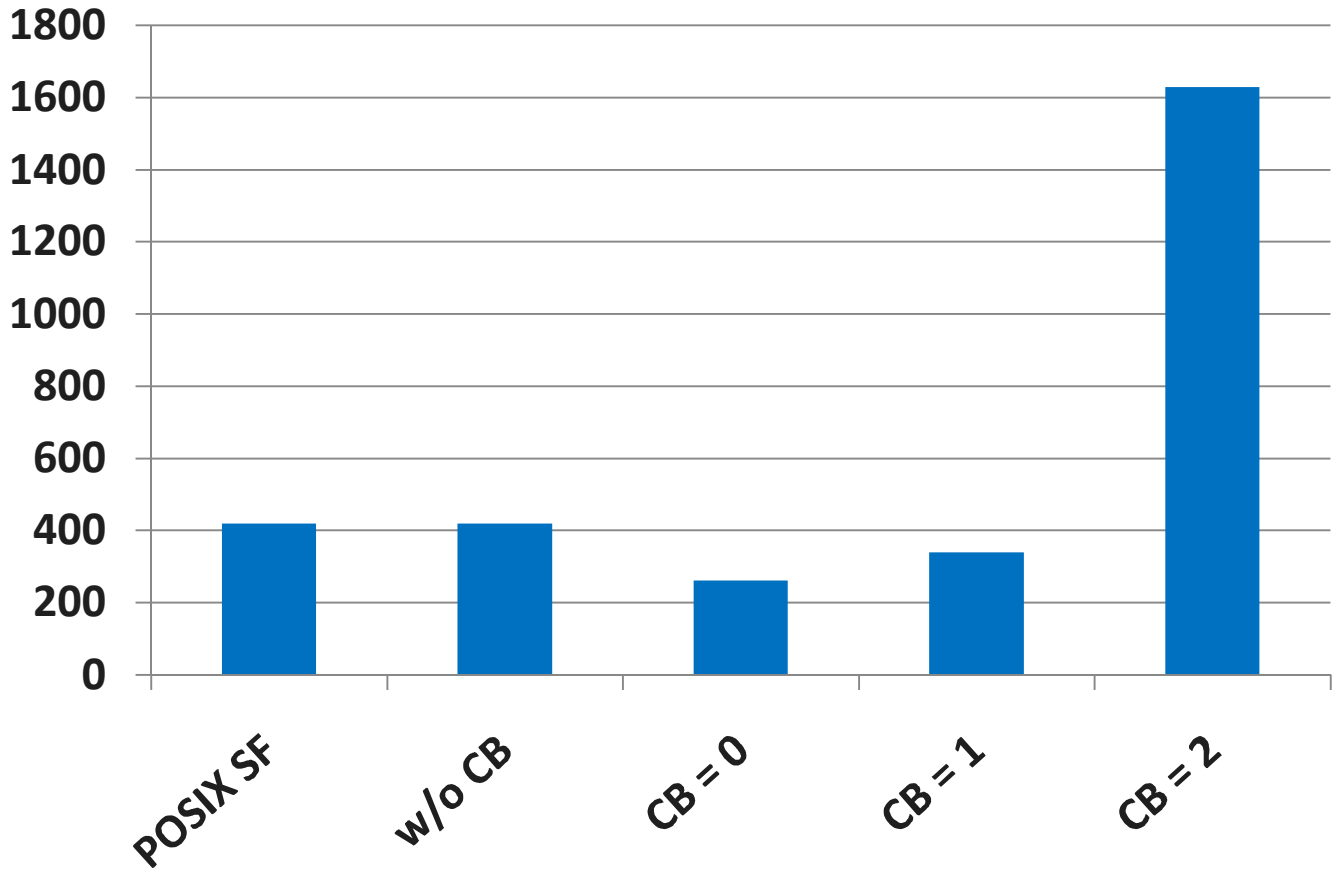
direct_io	cb_nodes	romio_ds_write
romio_cb_read	cb_config_list	ind_rd_buffer_size
romio_cb_write	romio_no_indep_rw	Ind_wr_buffer_size
cb_buffer_size	romio_ds_read	striping_factor
		striping_unit

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

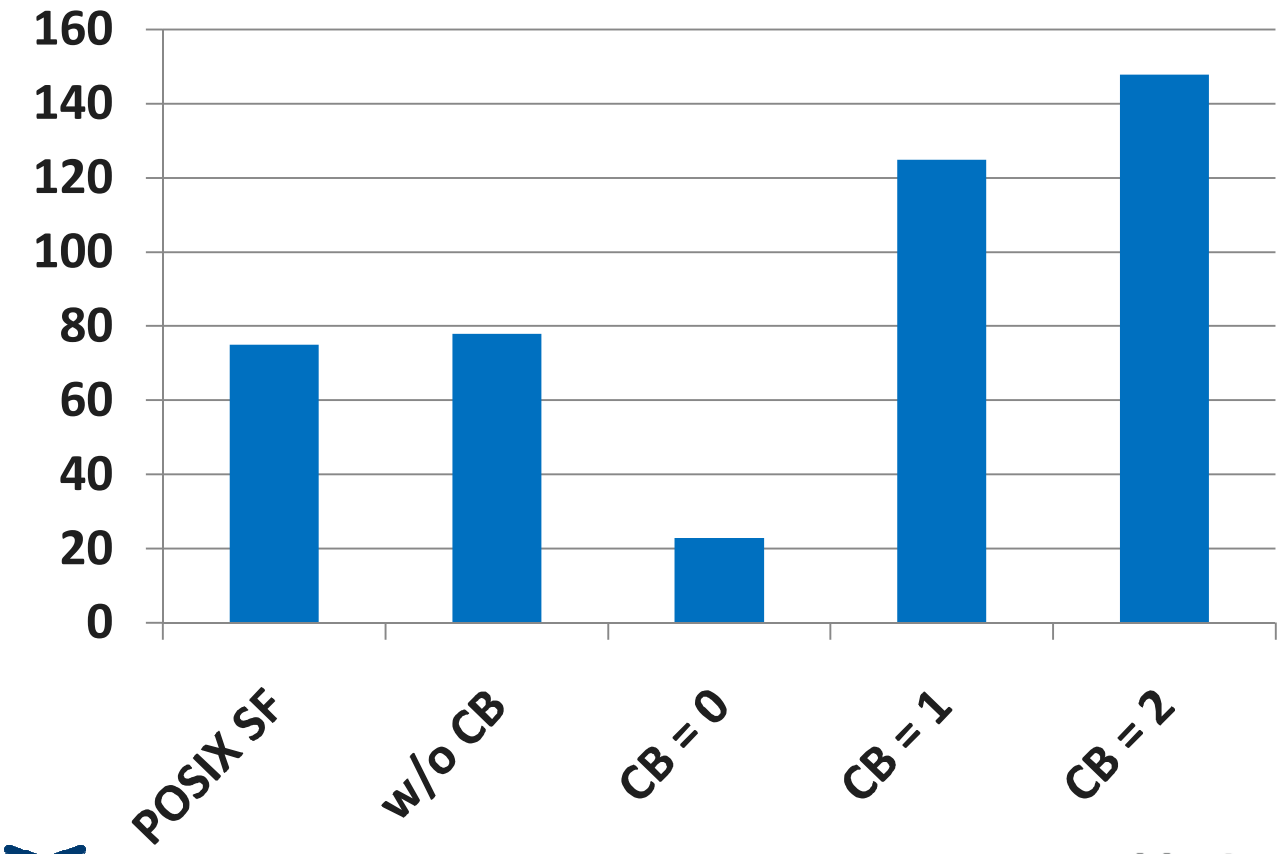


CRAY THE SUPERCOMPUTER COMPANY

HLRS

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

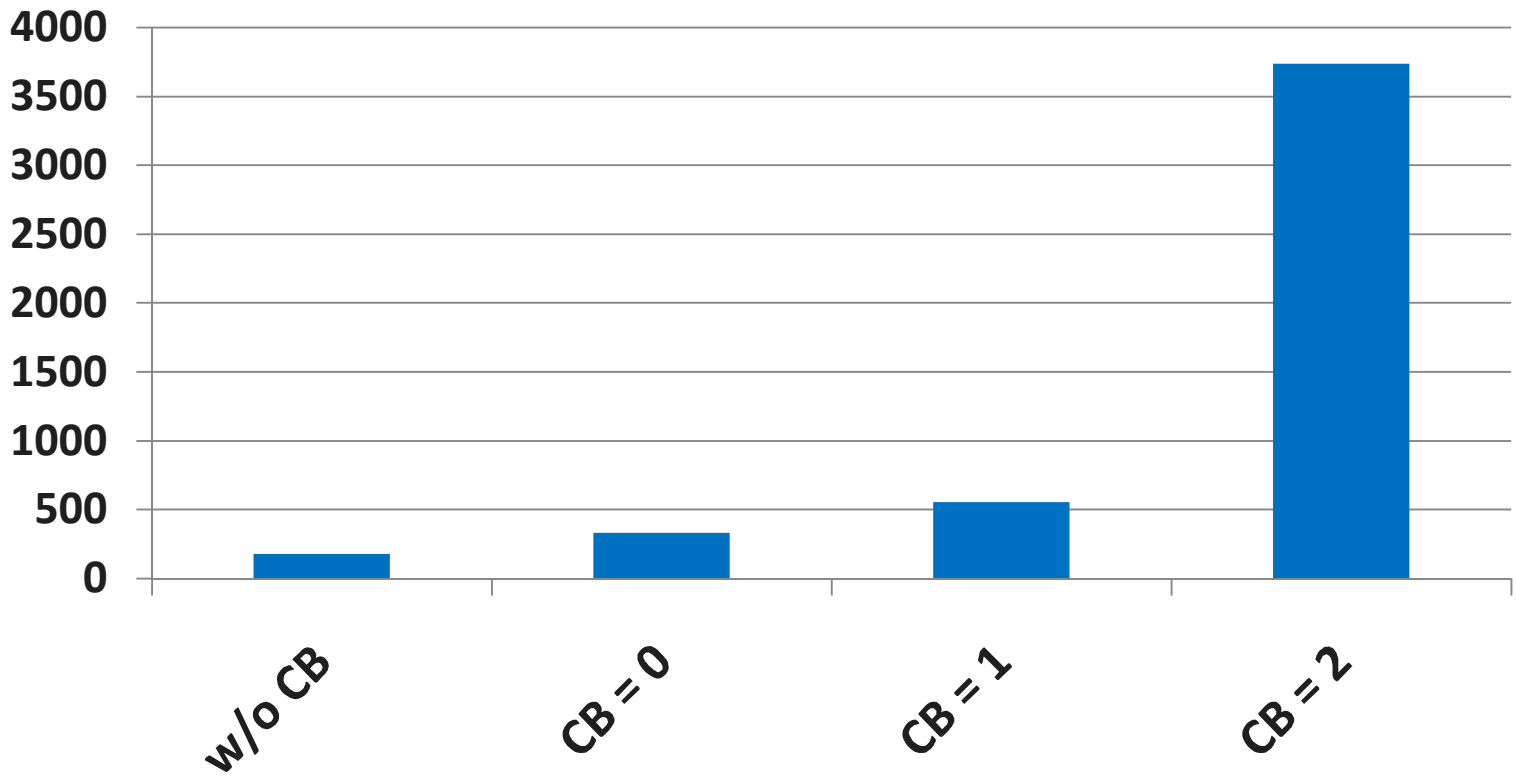


CRAY THE SUPERCOMPUTER COMPANY

HLRS

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

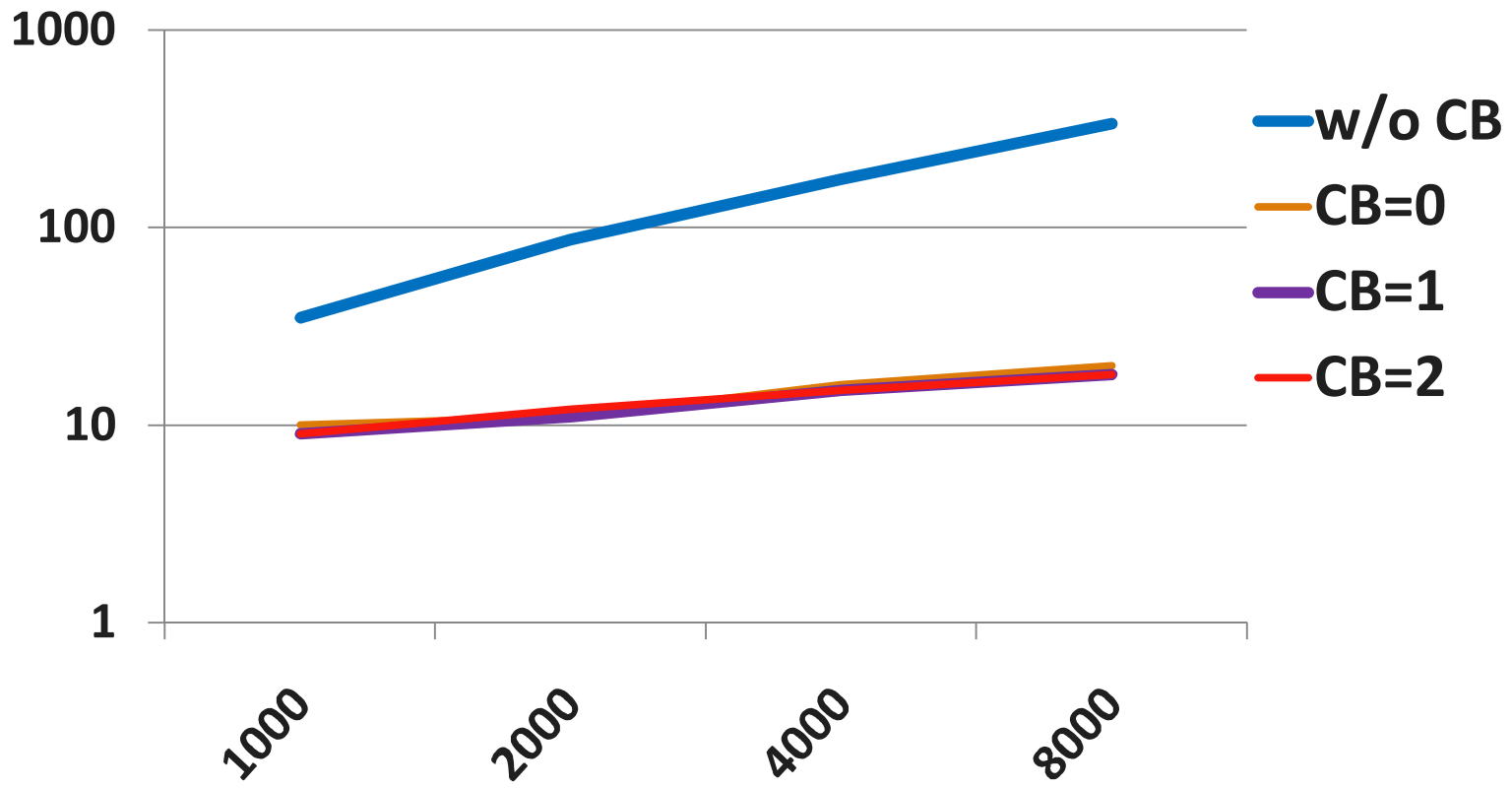


CRAY THE SUPERCOMPUTER COMPANY

HLRS

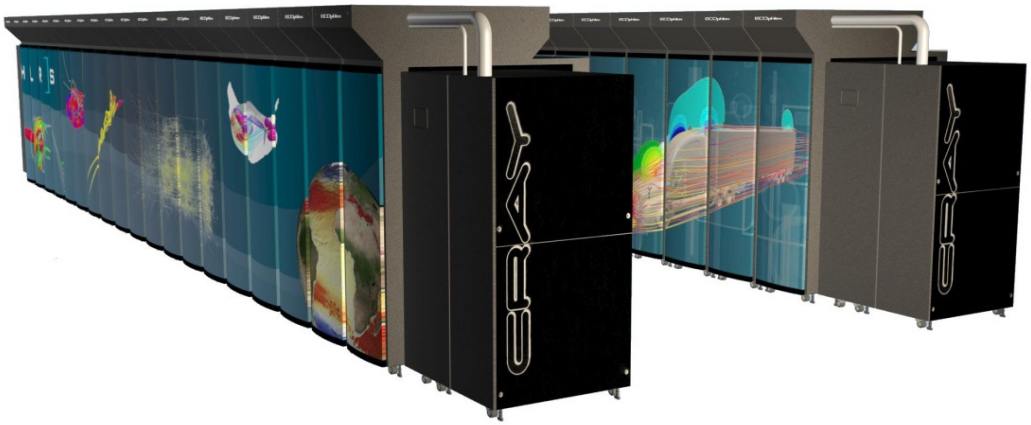
HDF5 format dump file from all PEs

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



CRAY THE SUPERCOMPUTER COMPANY

HLRS



MPI-IO Example

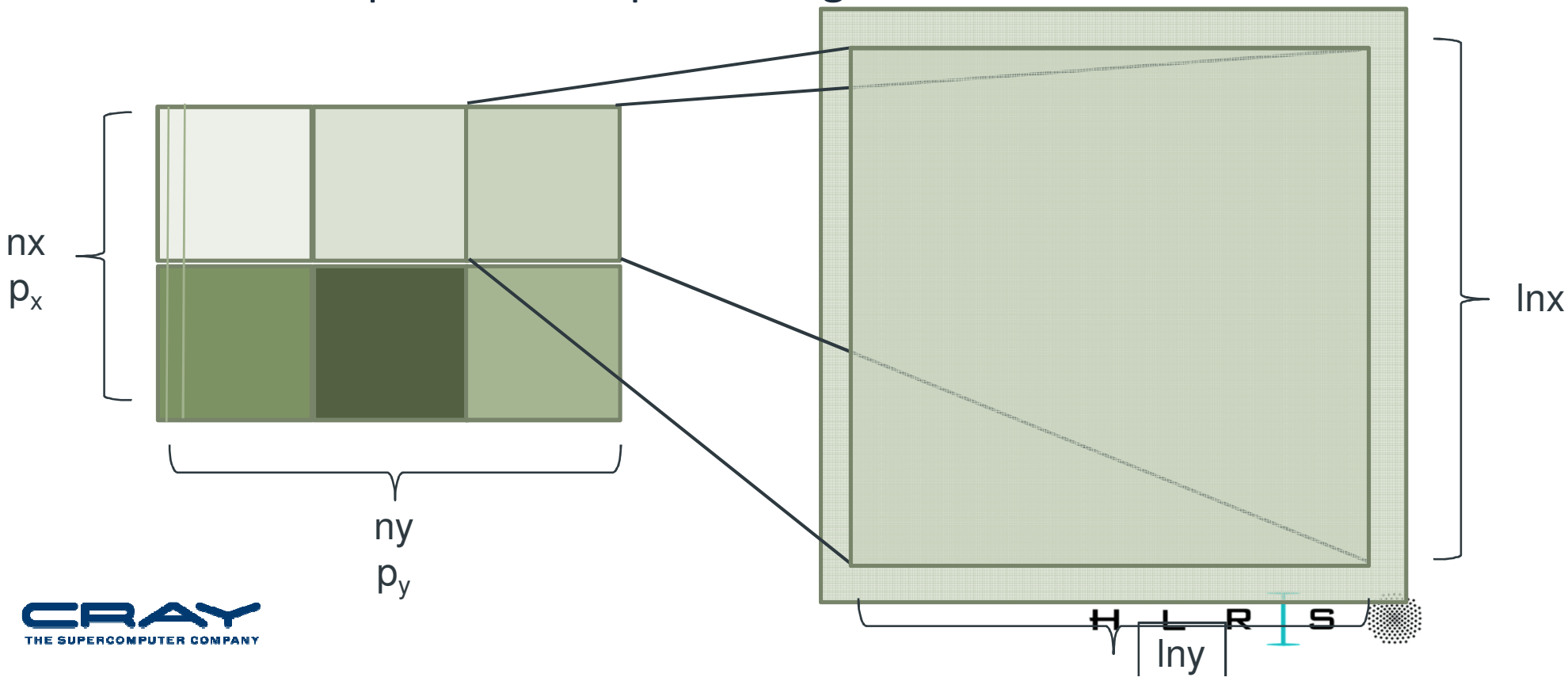
Storing a distributed Domain into a single File

CRAY THE SUPERCOMPUTER COMPANY

HLRS

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 :



CRAY
THE SUPERCOMPUTER COMPANY

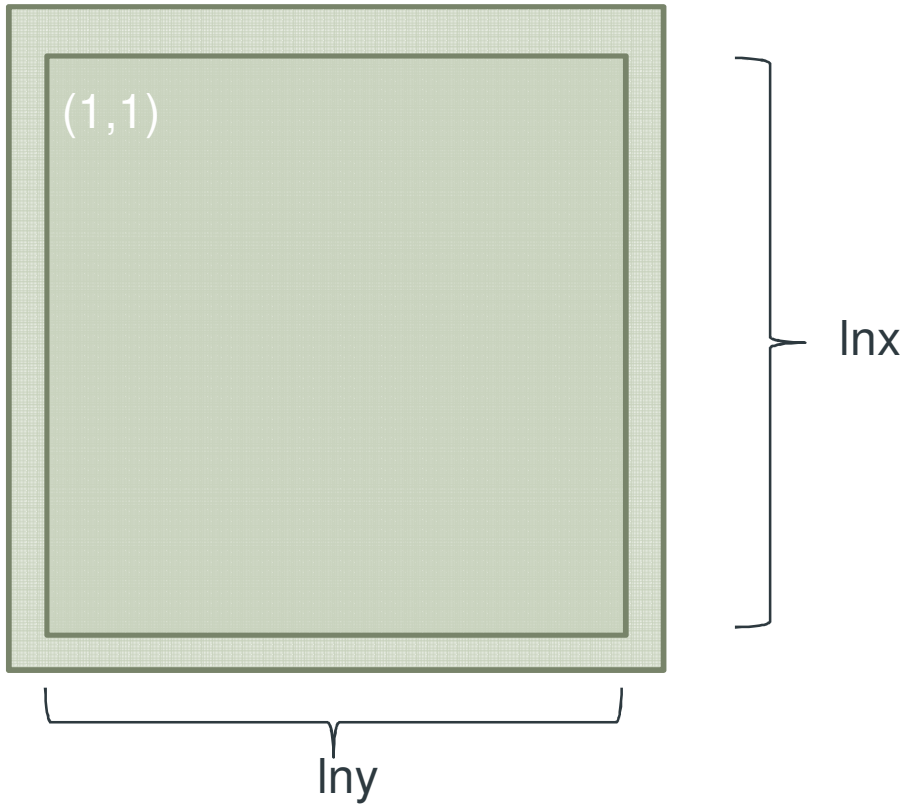
Approach for writing the file

- 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

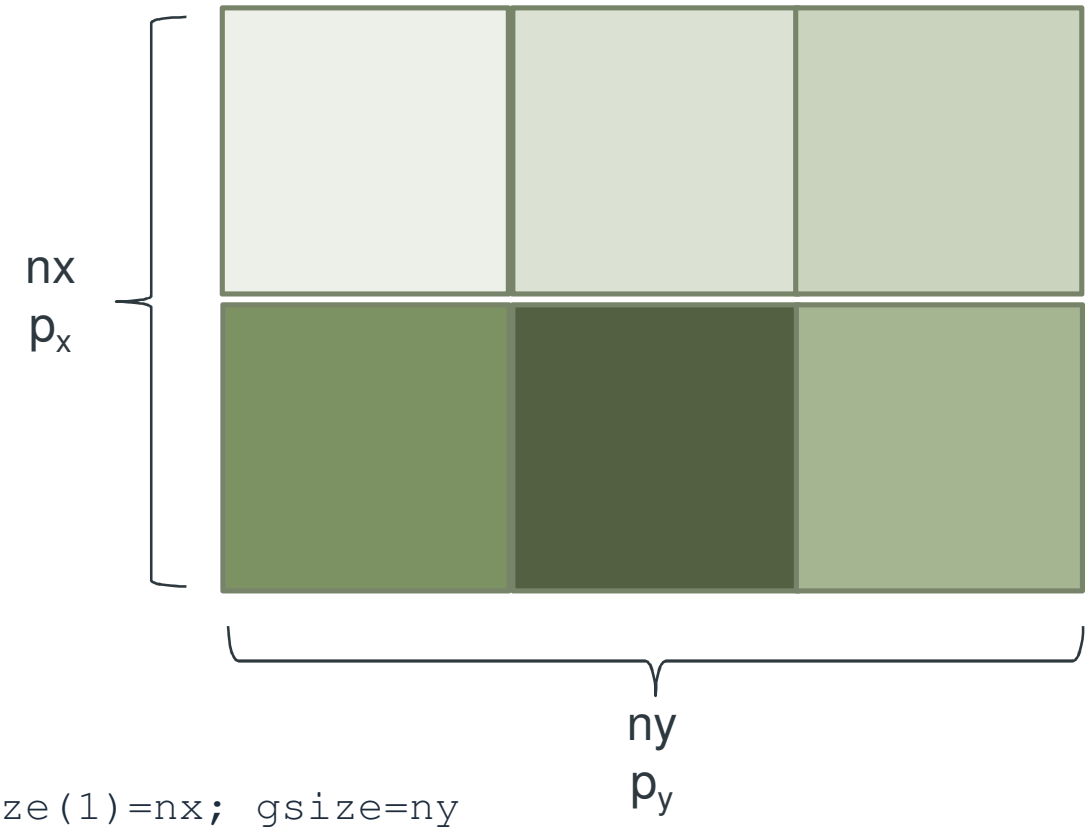
```
nx=512; ny=512 ! Global Domain Size
call MPI_Init(mpierr)
call MPI_Comm_size(MPI_COMM_WORLD, mysize, mpierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, mpierr)

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)
```

Performance results for the 2D testcase

- Global Domainsize = 4096x4096
- 16 MPI tasks
- 8 OSTs lustre filesystem
- System was not dedicated
- Timing includes HINT/WRITE not OPEN/CLOSE

	Performance in MB/sec for 16 MPI tasks			
	16 nodes -lmppnppn=1	8 nodes -lmppnppn=2	4 nodes -lmppnppn=4	2 nodes -lmppnppn=8
MPICH_MPIIO CB_ALIGN=	2 OSTs /8 OSTs	2 OSTs /8 OSTs	2 OSTs /8 OSTs	2 OSTs /8 OSTs
Unset	1523/1236	903/801	390/477	380/242
0	1513/1257	924/871	400/488	382/244
1	1509/1262	909/832	402/492	377/250
2	808/2100	780/1868	778/1501	738/872

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/


JAGUAR
APPLIED MECHANICS
Railway Technical Research Institute
H L R I S
CRAY XT5
CRAY XT5
Thank You!