Optimising I/O on the Cray XE6

PDC/KTH Performance Workshop
12-14 Sep 2012

Primary File Systems on lindgren

- **There are two primary types of filesystem on lindgren**

Home Space

- At /afs/pdc.kth.se/home/… (AFS)
- Smaller size (GBs)
- Only available on the login nodes

Designed and optimised for compilation, editing, long term storage of critical files.

Work Space

- Mounted as /cfs/klemming (Lustre)
- Split to scratch and nobackup
- Large size (285TB total)
- **IT IS NOT BACKED UP**
- Available on the compute and login nodes.
- Stage host **cfs-aux-4.pdc.kth.se**

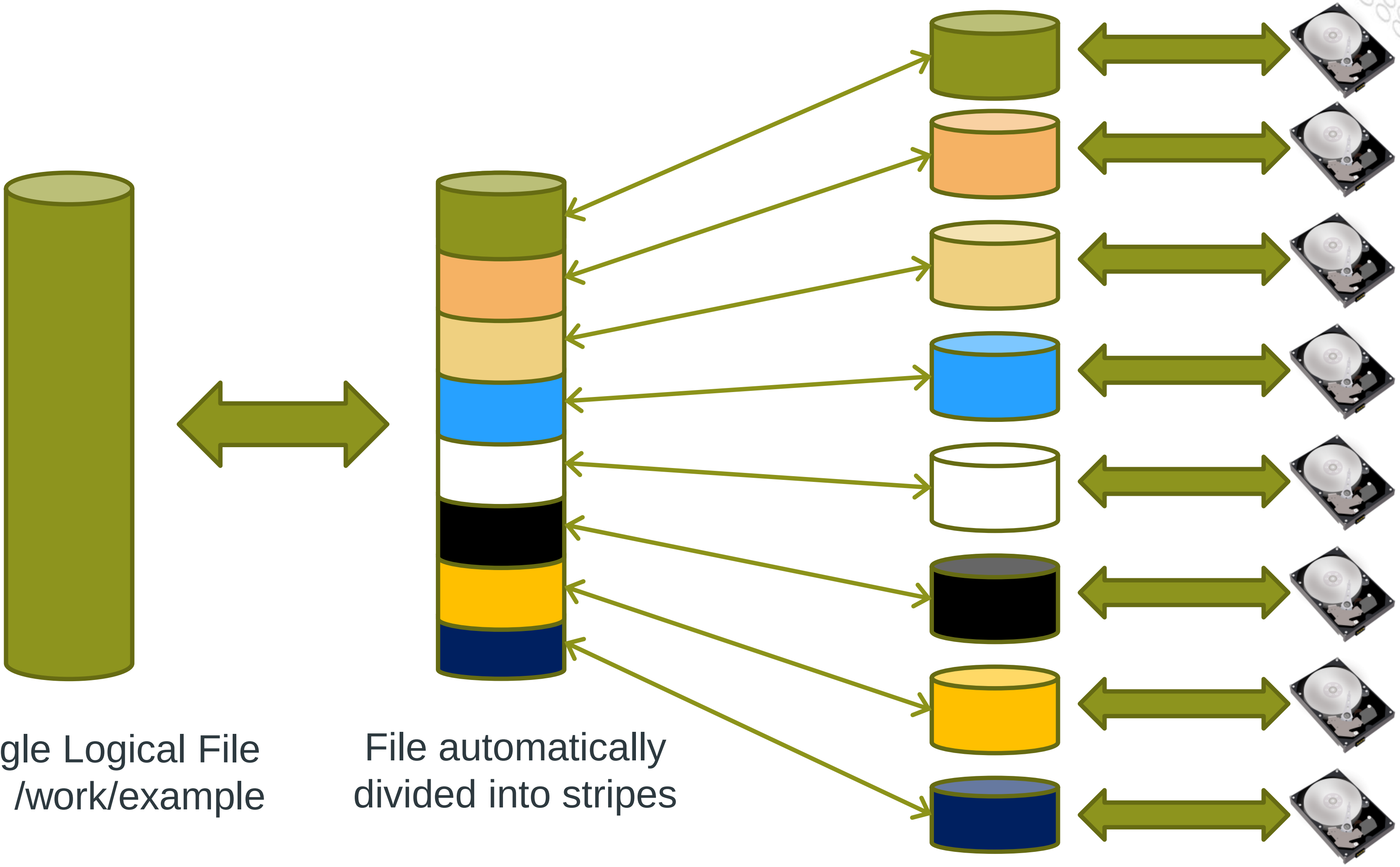
Designed and optimised for scratch large files and high bandwidth transfers (e.g. scientific output, restart files)

There are no local disks on the compute nodes

Understanding Parallel Filesystems

Concepts for reading or writing files to lustre

File System Fundamentals

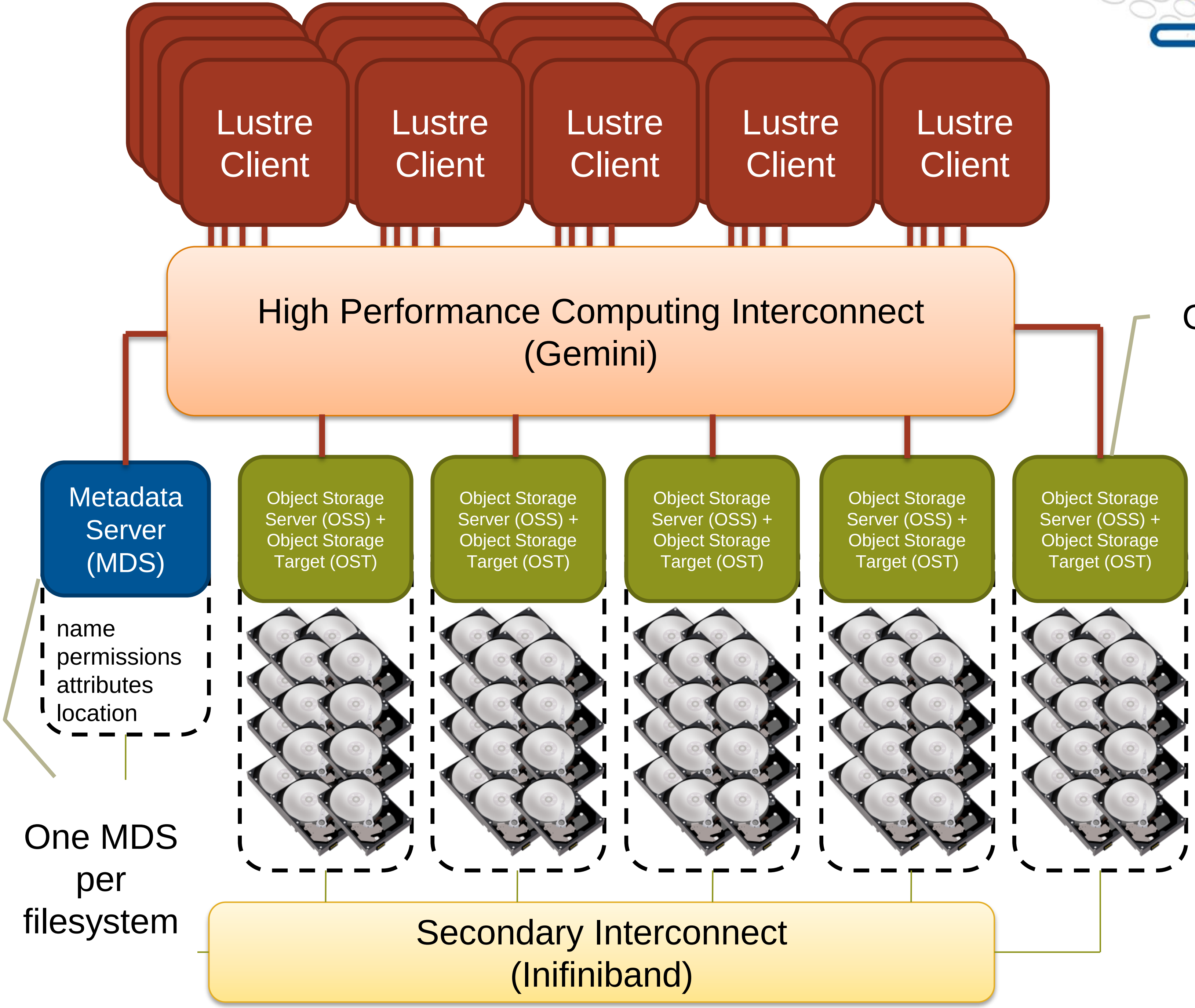


Single Logical File
e.g. /work/example

File automatically
divided into stripes

Stripes are written/read
from across multiple drives

- **A scalable cluster file system for Linux**
  - Developed by Cluster File Systems -> Sun -> Oracle.
  - Name derives from “Linux Cluster”
  - The Lustre file system consists of software subsystems, storage, and an associated network
- **MDS – metadata server**
  - Handles information about files and directories
- **OSS – Object Storage Server**
  - The hardware entity
  - The server node
  - Support multiple OSTs
- **OST – Object Storage Target**
  - The software entity
  - This is the software interface to the backend volume



Lustre Client

Lustre Client

Lustre Client

Lustre Client

Lustre Client

High Performance Computing Interconnect (Gemini)

Metadata Server (MDS)

- name
- permissions
- attributes
- location

Object Storage Server (OSS) + Object Storage Target (OST)

Object Storage Server (OSS) + Object Storage Target (OST)

Object Storage Server (OSS) + Object Storage Target (OST)

Object Storage Server (OSS) + Object Storage Target (OST)

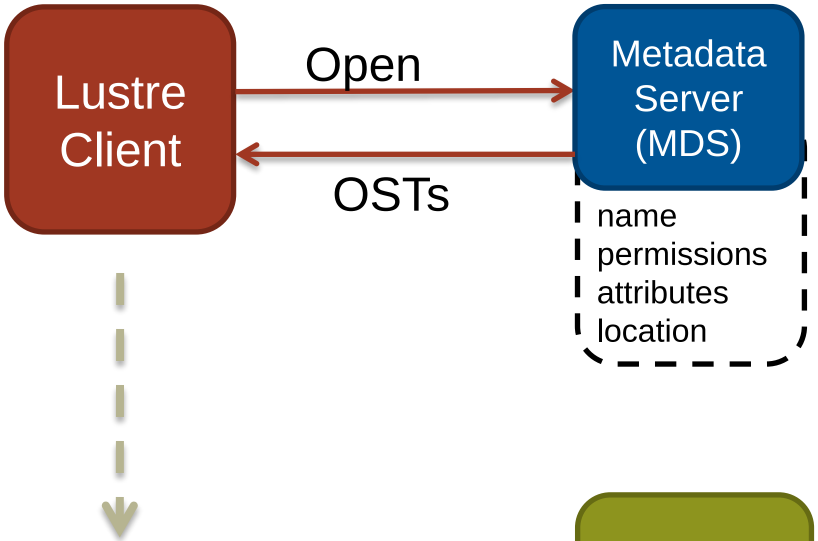
Object Storage Server (OSS) + Object Storage Target (OST)

Secondary Interconnect (Inifiniband)

Multiple OSSs and OSTS

One MDS per filesystem

Opening a file


Lustre Client

Open

Metadata Server (MDS)

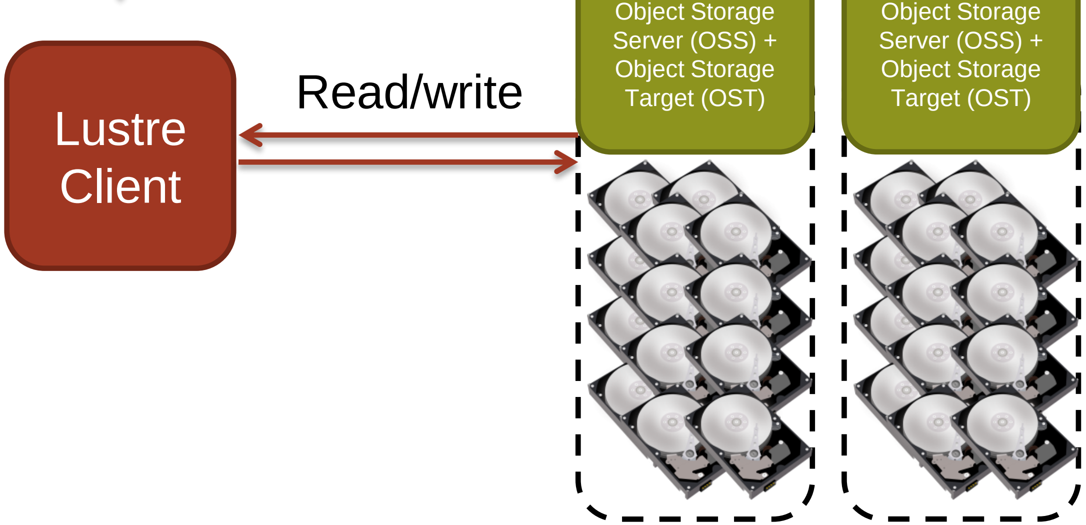
OSTs

name
permissions
attributes
location

The client sends a request to the MDS to opening/acquiring information about the file

The MDS then passes back a list of OSTs

- For an existing file, these contain the data stripes
- For a new files, these typically contain a randomly assigned list of OSTs where data is to be stored


Lustre Client

Read/write

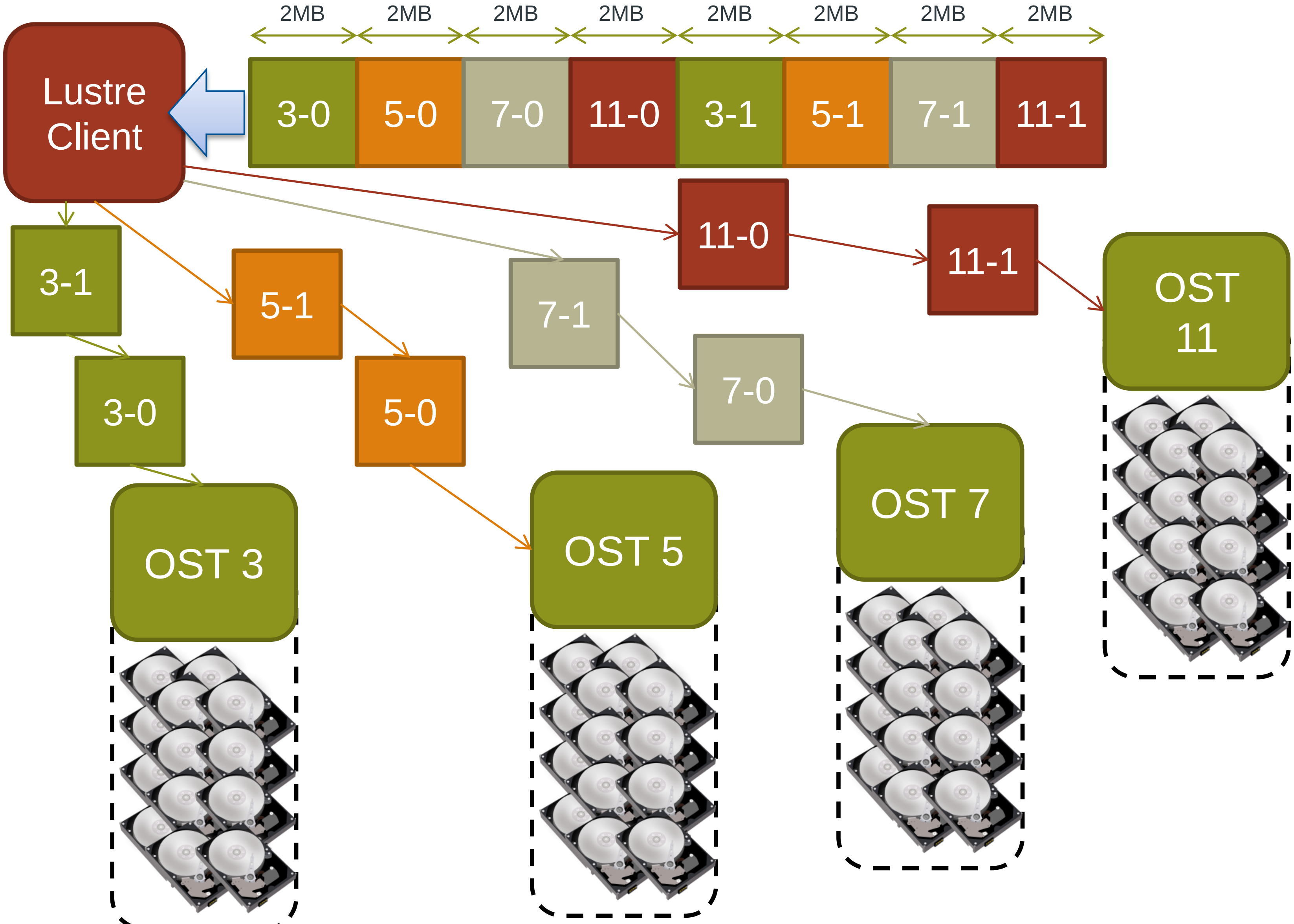
Object Storage Server (OSS) + Object Storage Target (OST)

Object Storage Server (OSS) + Object Storage Target (OST)

Once a file has been opened no further communication is required between the client and the MDS

All transfer is directly between the assigned OSTs and the client

File decomposition – 2 Megabyte Stripes



Controlling Lustre Striping

- `lfs` - the lustre utility for setting the stripe properties of new files, or displaying the striping patterns of existing.

- The most used options are :
  - `setstripe` – Set striping properties of a directory or new file
  - `getstripe` – Return information on current striping settings
  - `osts` – List the number of OSTs associated with this file system
  - `df` – Show disk usage of this file system

- For help execute lfs without any arguments

```
$ lfs
lfs > help
Available commands are:
        setstripe
        find
        getstripe
        check
       ……….
```

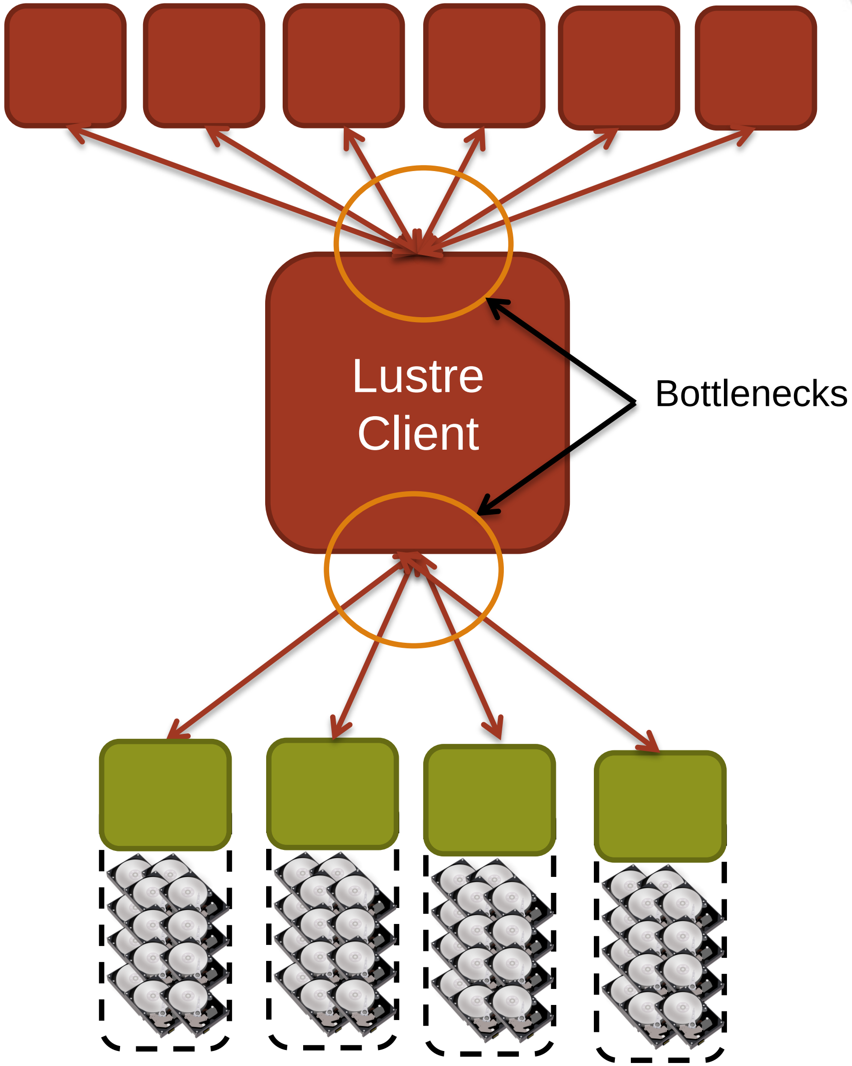
lfs setstripe

- **Sets the stripe for a file or a directory**
- `lfs setstripe <file|dir> <-s size> <-i start> <-c count>`
  - size: Number of bytes on each OST (0 filesystem default)
  - start: OST index of first stripe (-1 filesystem default)
  - count: Number of OSTs to stripe over (0 default, -1 all)
- **Comments**
  - Can use lfs to create an empty file with the stripes you want (like the touch command)
  - Can apply striping settings to a directory, any children will inherit parent’s stripe settings on creation.
  - The stripes of a file is given when the file is created. It is not possible to change it afterwards.

Common I/O Paradigms

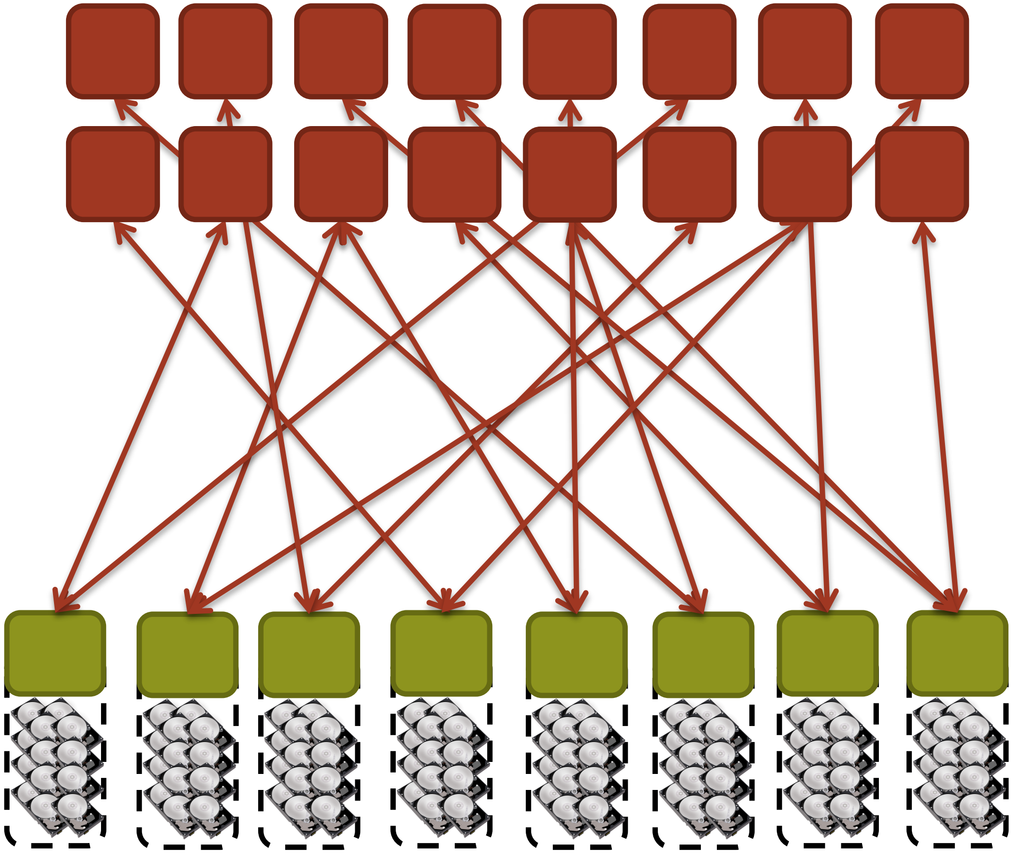
Spokesperson

- **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)**



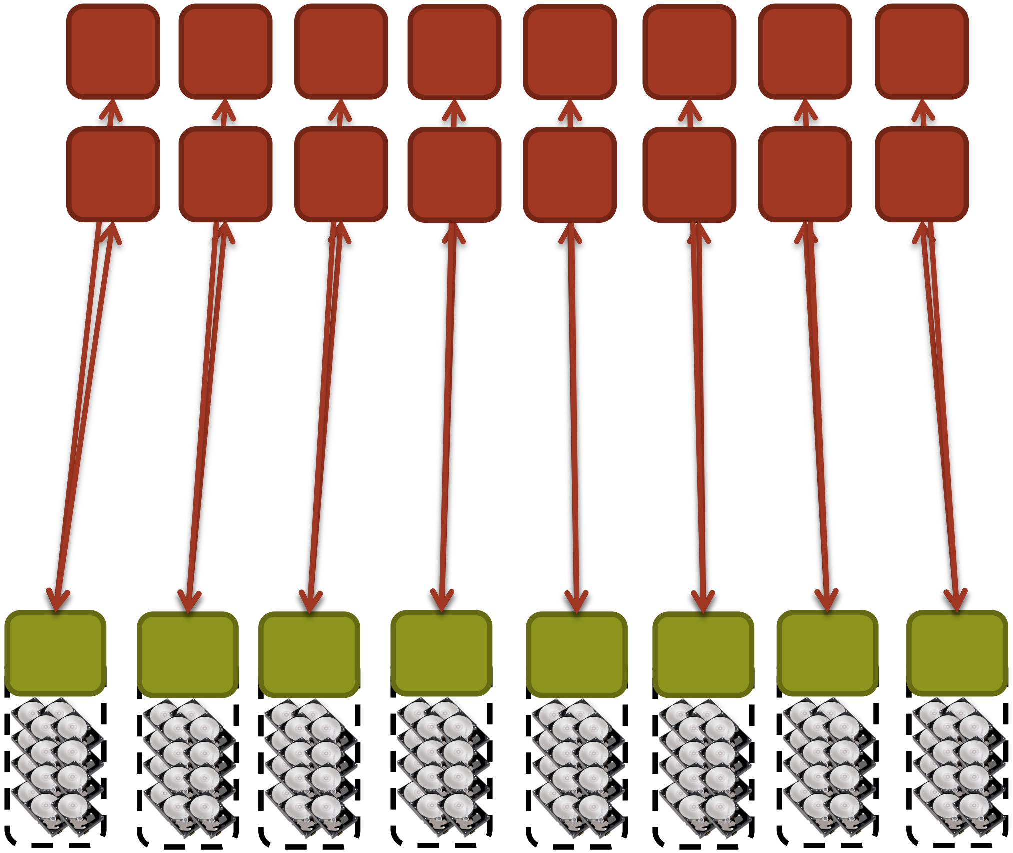
Multiple Writers – Multiple Files

- **All processes perform I/O to individual files.**
  - Limited by file system.
- **Easy to program**
  - Requires job to always run on the same number of cores
- **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.



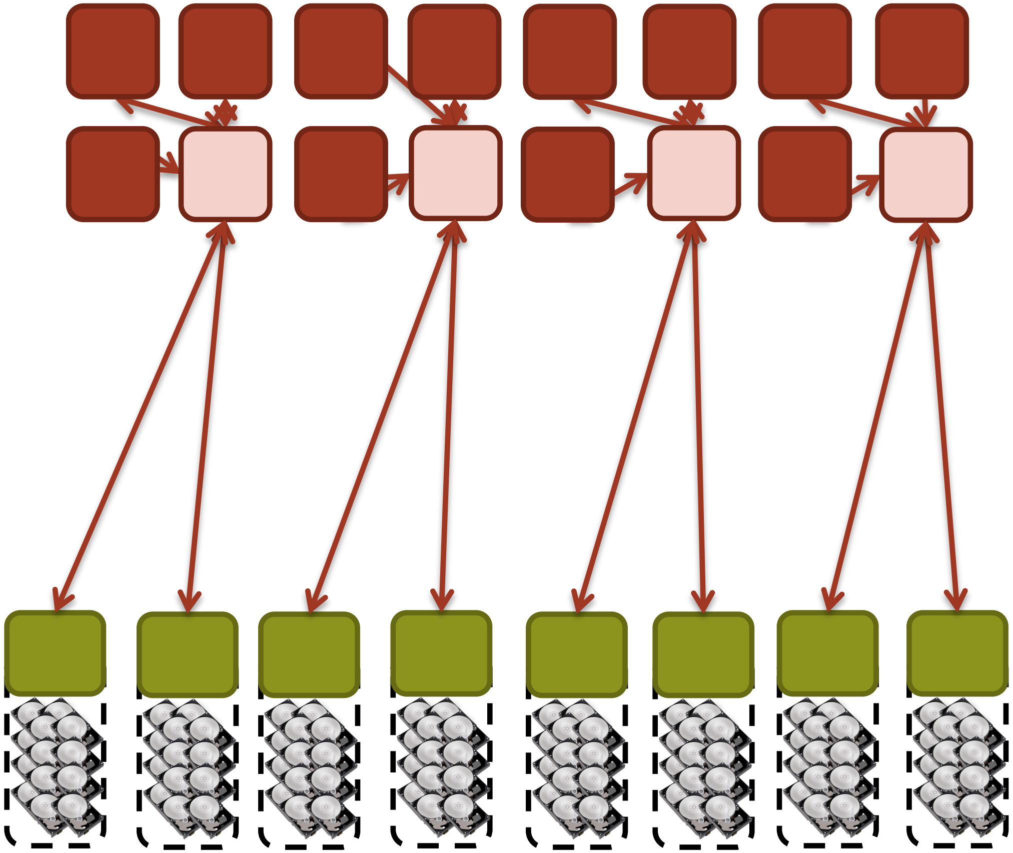
Multiple Writers – Single File

- **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.
- **Not all programming languages support it**
  - C/C++ can work with fseek
  - No real Fortran standard



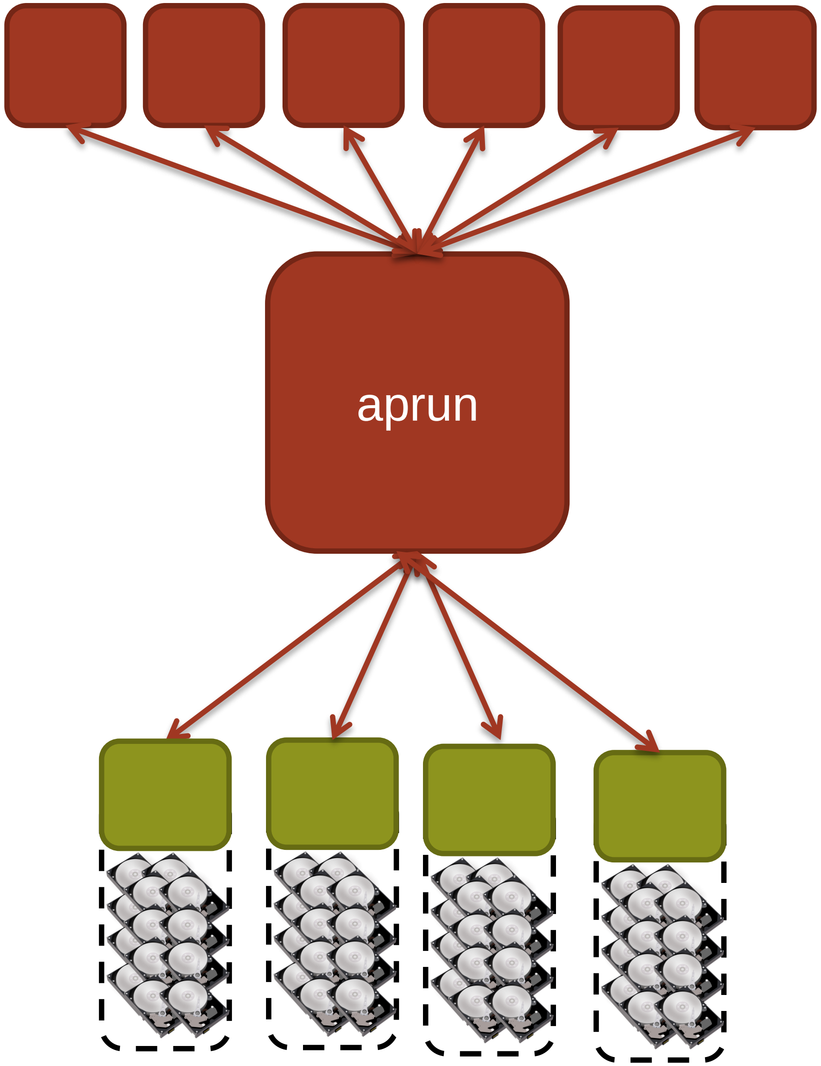
Collective IO to single or multiple files

- **Aggregation to a processor in a 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.

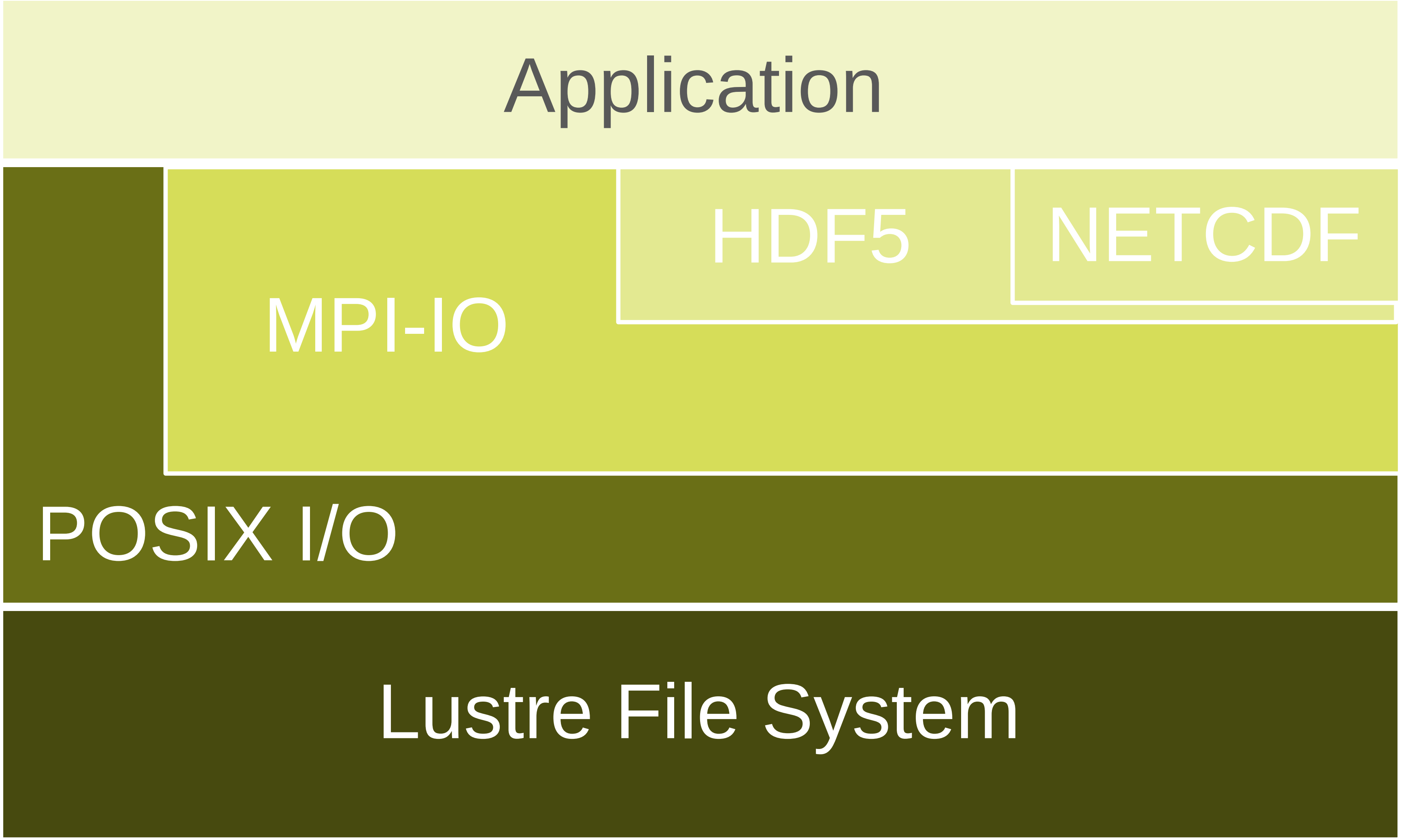


Special Case : Standard Output and Error

- 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 32,000!"
  - "Task 64,000, made it through loop."
  - ...



CRAY IO Software stack



Application

HDF5

NETCDF

MPI-IO

POSIX I/O

Lustre File System

Optimising I/O in Applications

1. Tune your parameters

Any easy and non-invasive approach

Select best striping values

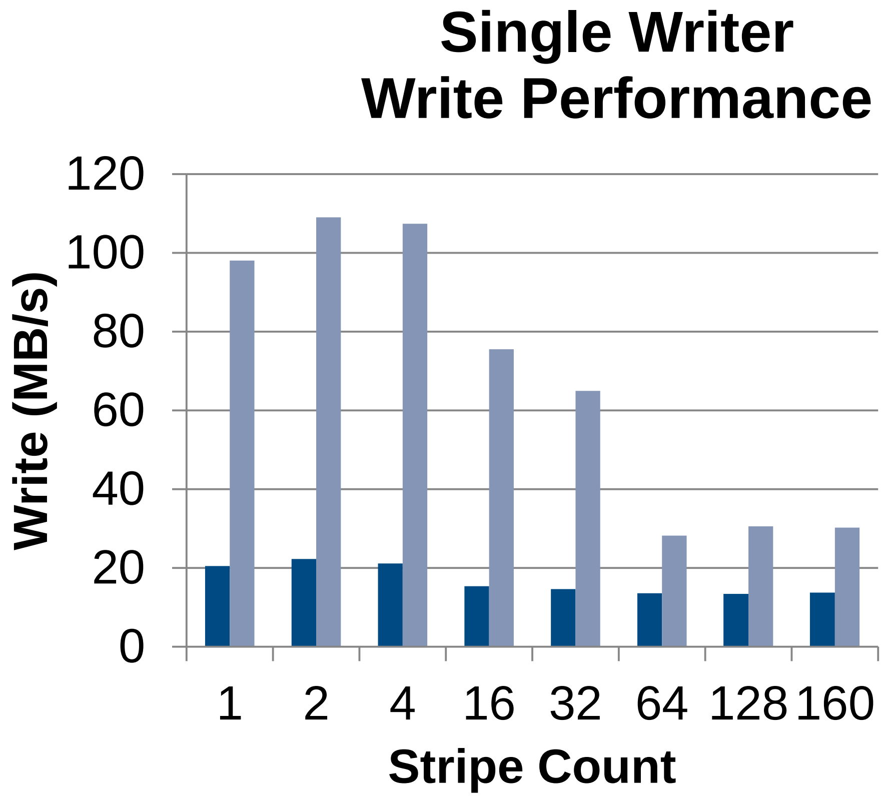
- **Selecting the striping values will have an impact on the I/O performance of your application**
- **Rule of thumb :**
  1. # files > # OSTs => Set stripe_count=1
     You will reduce the lustre contension and OST file locking this way and gain performance
  2. #files==1 => Set stripe_count=#OSTs
     Assuming you have more than 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

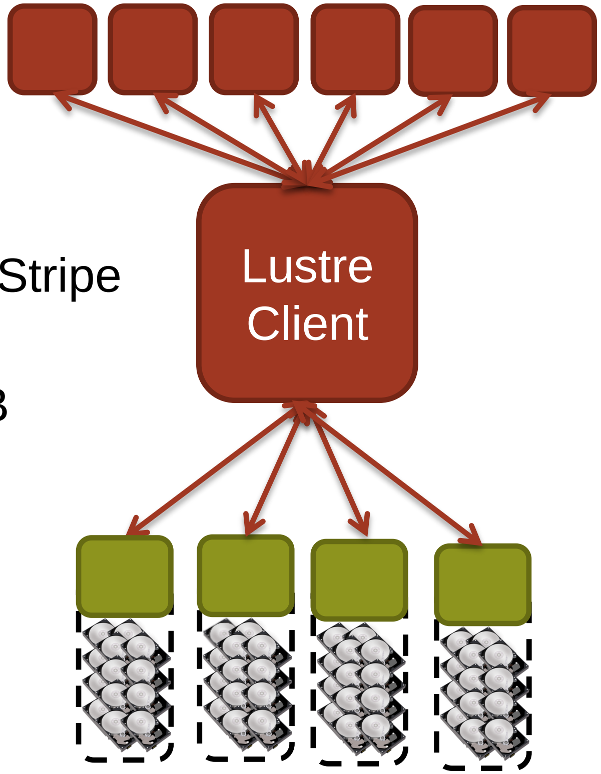
Always allow the system to choose OSTs at random!

(Ensures even loading of the OSTs and prevents accidental contention)

Case Study 1 : Spokesman

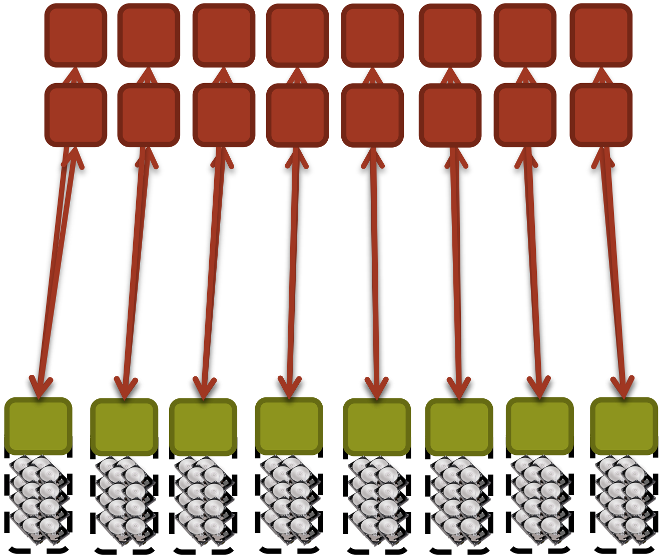
- **32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size**
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance
  - Note : XE6 numbers might be better





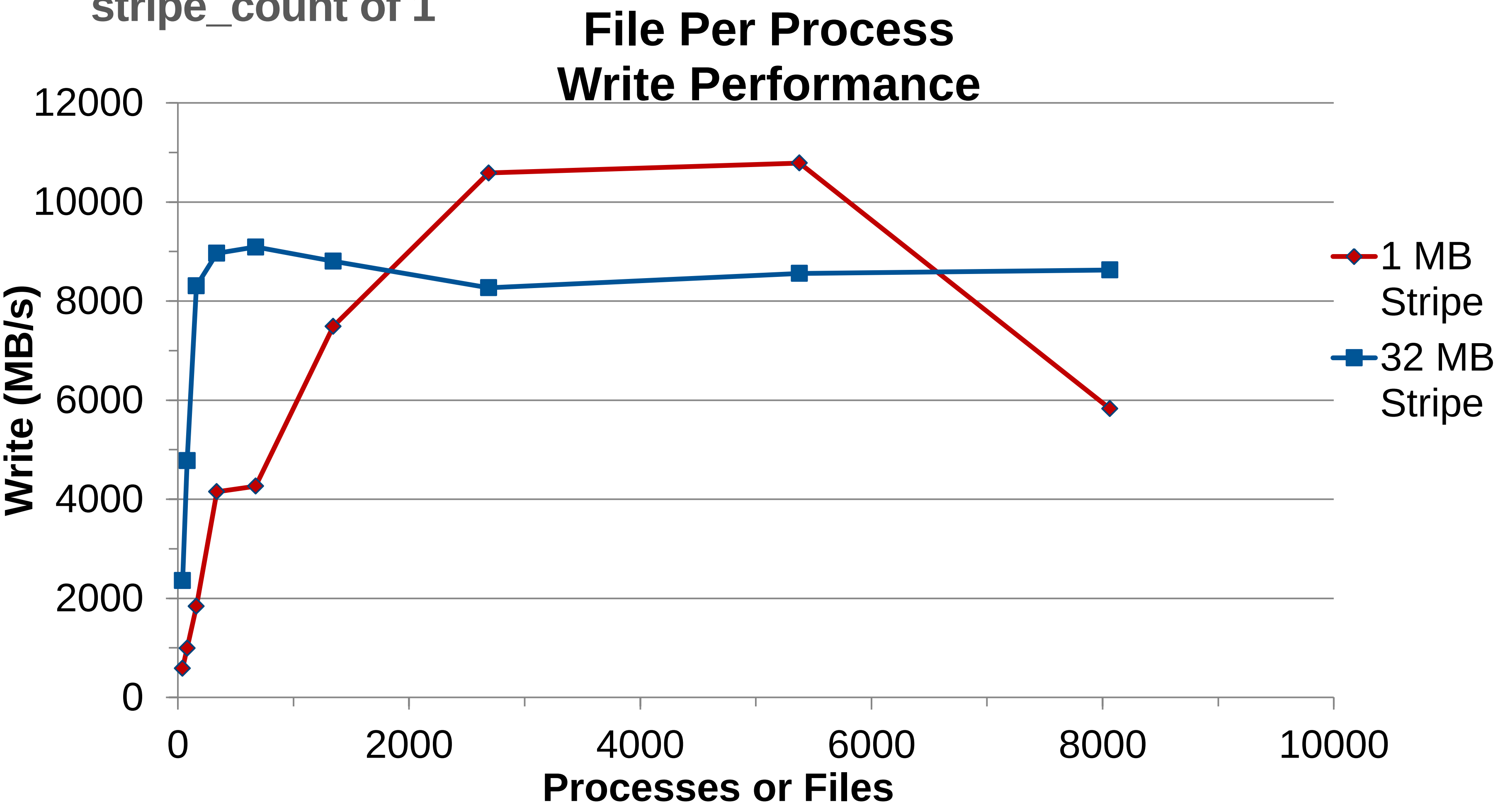
Case Study 2 : Parallel I/O into a single file

- **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



2. Try to hide the I/O

Asynchronous I/O

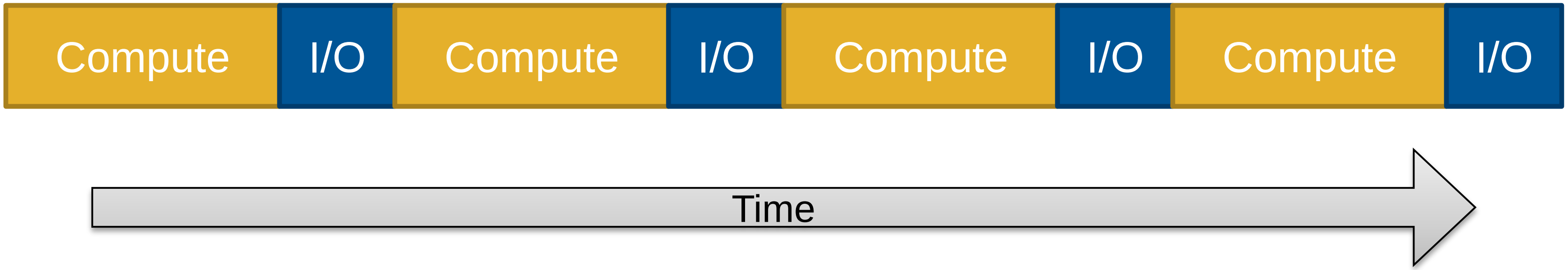
Majority of data is output, allow computation to overlap

Double buffer arrays to allow computation to continue while data flushed to disk

1. **Use asynchronous POSIX calls**
   - Only covers the I/O call itself, any packing/gathering/encoding still has to be done by the compute processors
   - Not currently supported by Lustre (calls become synchronous)
2. **Use 3rd party libraries**
   - Typical examples are MPI-IO (see later)
   - Again, packing/gathering/encoding still done by compute processors
3. **Add I/O Servers to the application**
   - Add processors dedicated to performing time consuming operations
   - More complicated to implement than other solutions
   - Portable across platforms (works on any parallel platform)

Asynchronous I/O

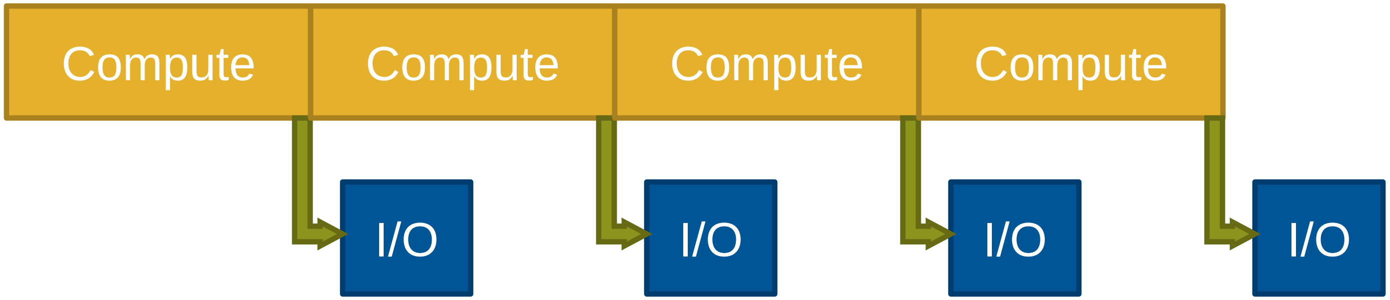
Standard Sequential I/O



Compute I/O Compute I/O Compute I/O Compute I/O

Time

Asynchronous I/O



Compute Compute Compute Compute

I/O I/O I/O I/O

Naive IO Server Pseudo Code

User more nodes to act as I/O Servers

Compute Node

```
do i=1,time_steps
  compute(j)
  checkpoint(data)
end do

subroutine checkpoint(data)
  MPI_Wait(send_req)
  buffer = data
  MPI_Isend(IO_SERVER, buffer)
end subroutine
```

I/O Server

```
do i=1,time_steps
  do j=1,compute_nodes
    MPI_Recv(j, buffer)
    write(buffer)
  end do
end do
```

IO Servers

- **Successful strategy deployed in multiple codes.**
- **Strategy has become more successful as number of nodes has increased.**
  - Addition of extra nodes only cost 1-2% in resources
- **Requires additional development that can pay off for codes that generate large files.**
- **Typically still only one or a small number of writers performing I/O operations (not necessarily reaching optimum bandwidth).**

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

3. Parallelise, e.g. MPI-IO

Change how the application handles I/O

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 safe (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

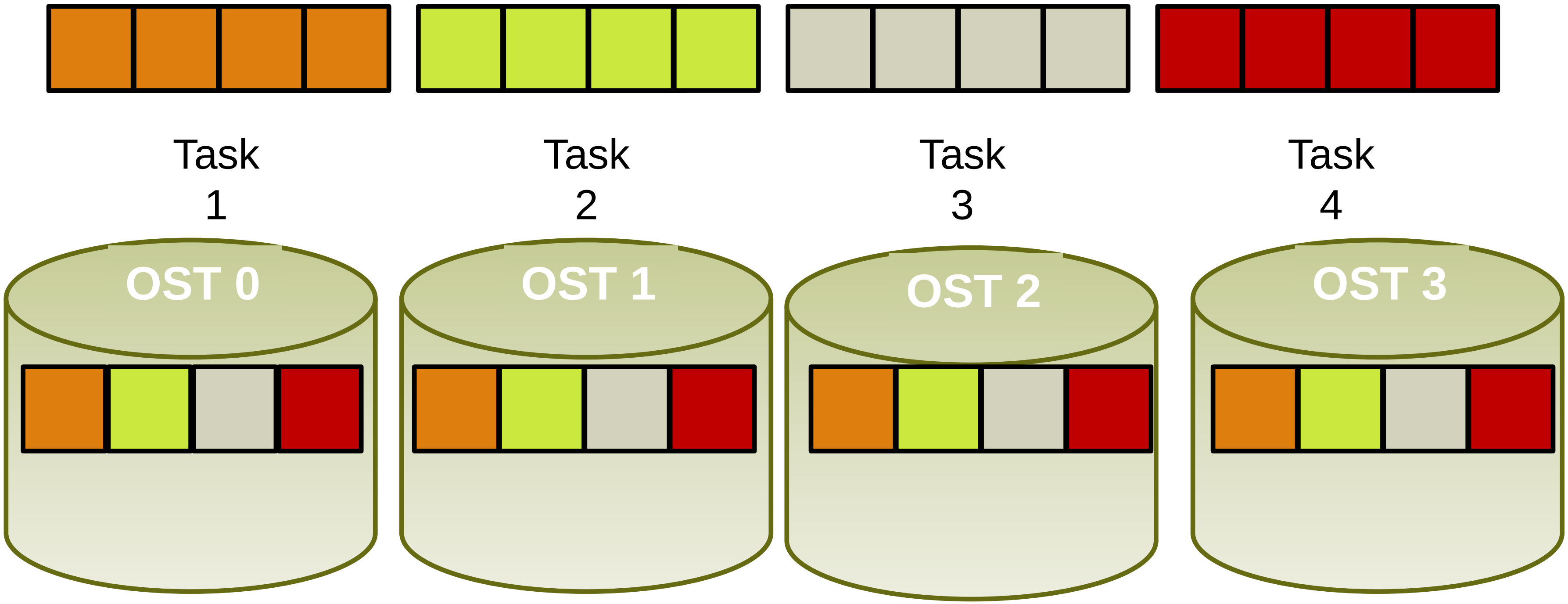
- **MPI_File_read_all, MPI_File_read_at_all, …**
- **_all indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function**
- **Each process specifies only its own access information – the argument list is the same as for the non-collective functions**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

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)
    - sets collective buffering behaviour
  - 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 romio_cb_read romio_cb_write cb_buffer_size	cb_nodes cb_config_list romio_no_indep_rw romio_ds_read	romio_ds_write ind_rd_buffer_size Ind_wr_buffer_size striping_factor striping_unit

Env. Variable MPICH_MPIO_HINTS (part 2)

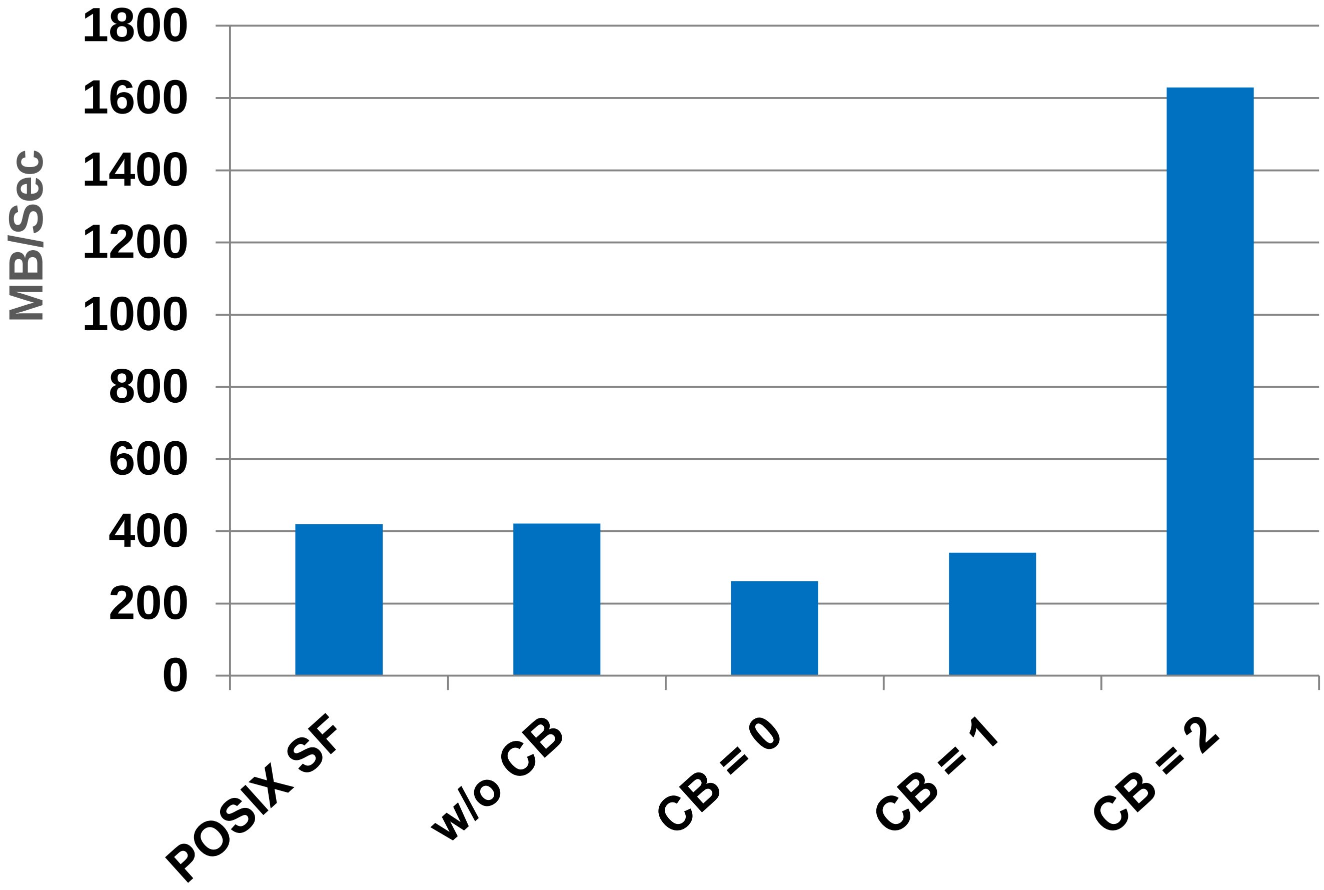
- If set, override the default value of one or more MPI I/O hints. This also overrides any values that were set by using calls to MPI_Info_set in the application code. The new values apply to the file the next time it is opened using a MPI_File_open() call.
- After the MPI_File_open() call, subsequent 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
  ```

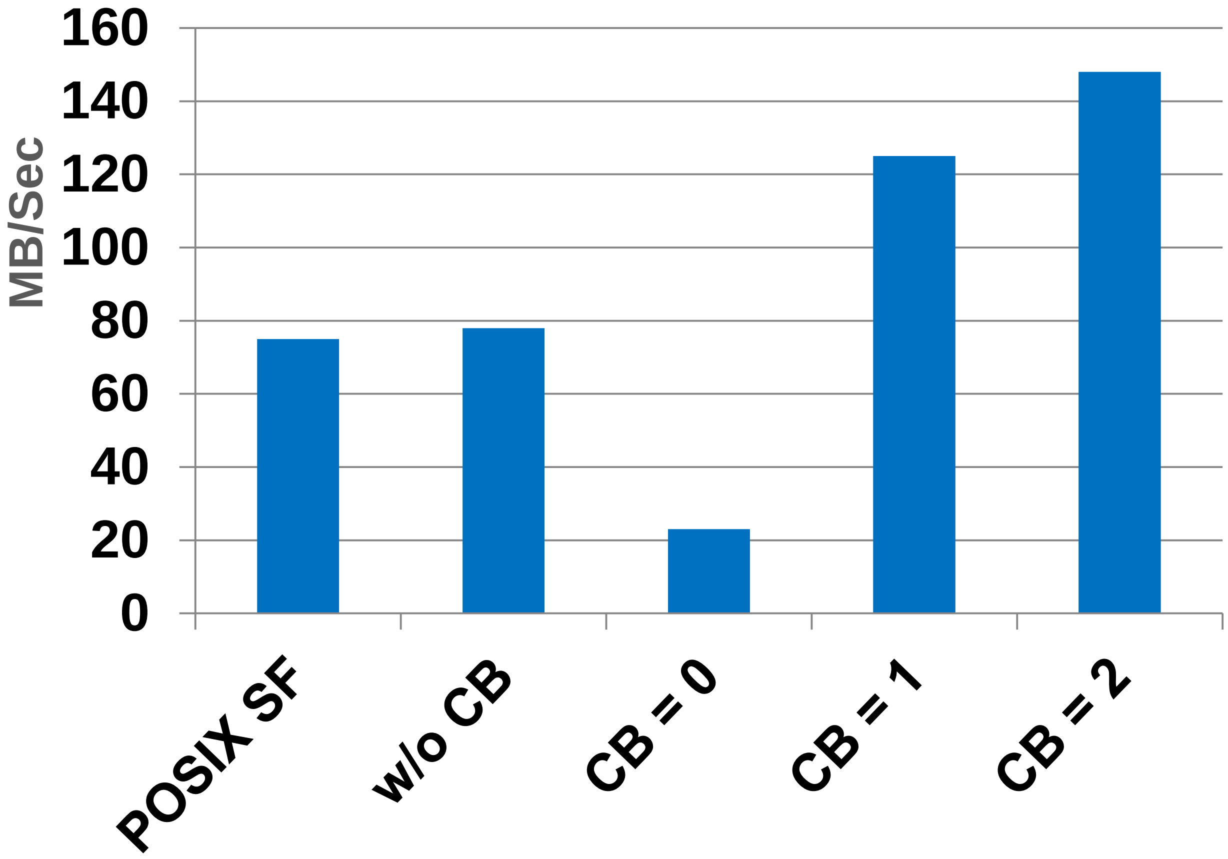
IOR benchmark 1,000,000 bytes

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



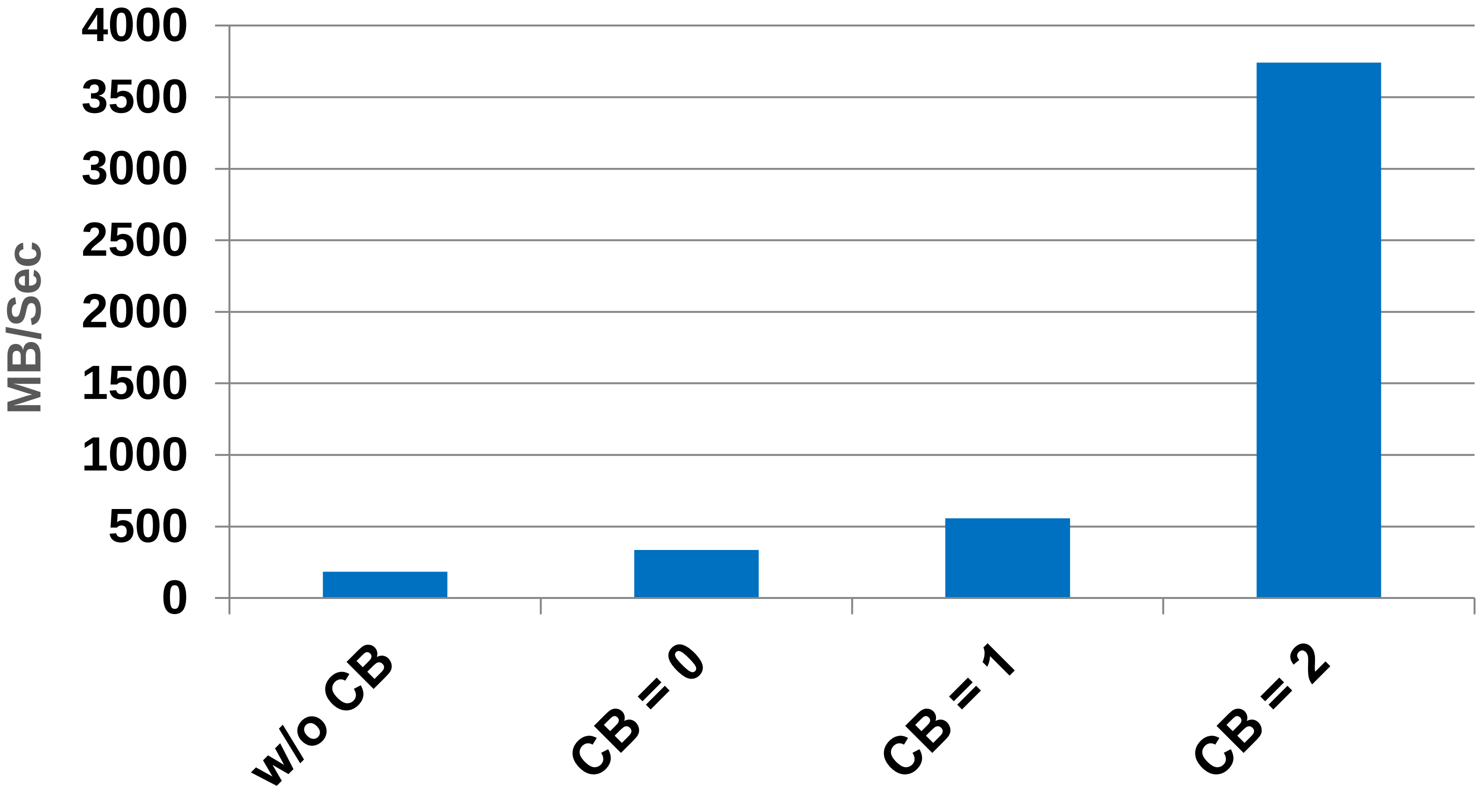
IOR benchmark 10,000 bytes

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



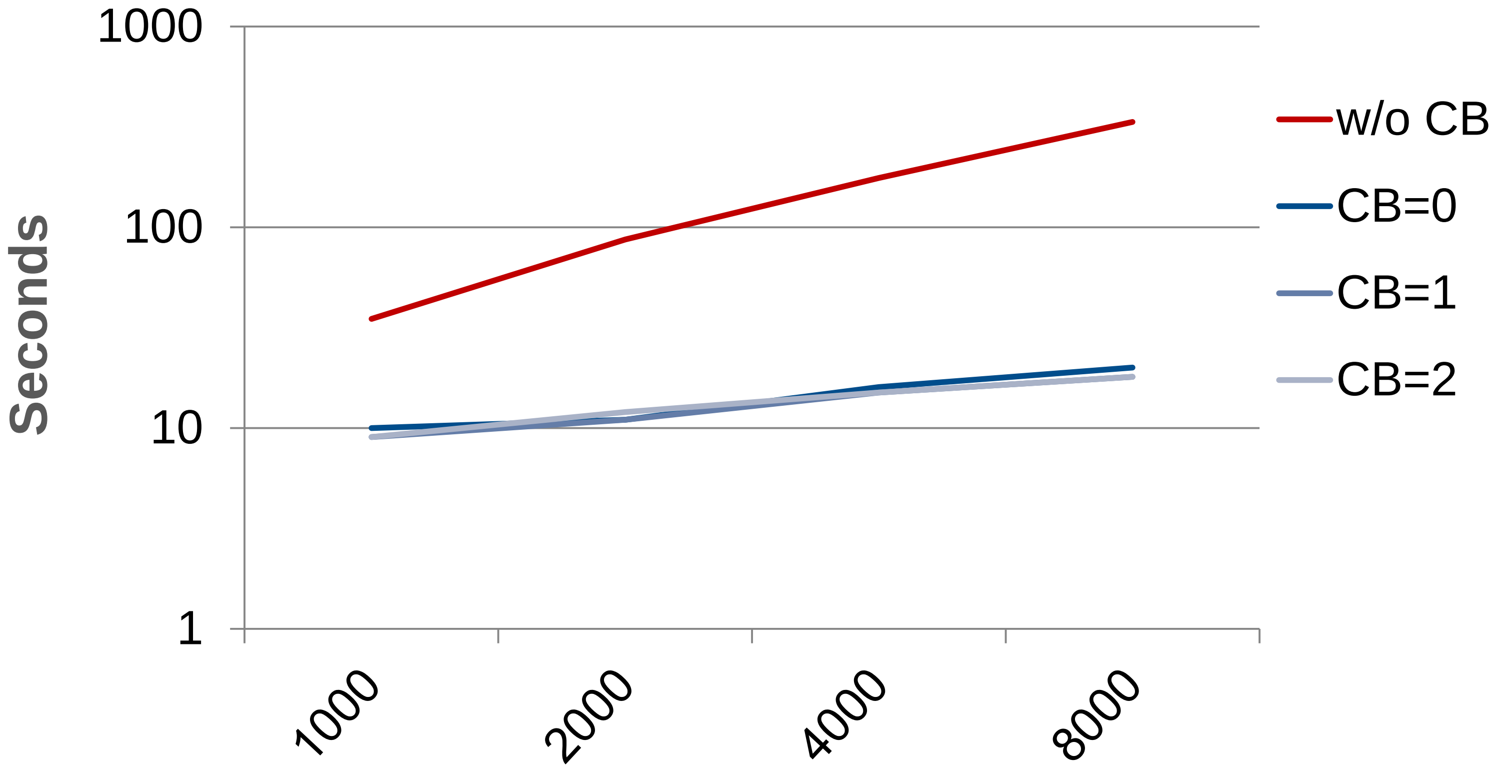
HYCOM MPI-2 I/O

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



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.
Tested on an XT5, 8 stripes, 8 cb_nodes

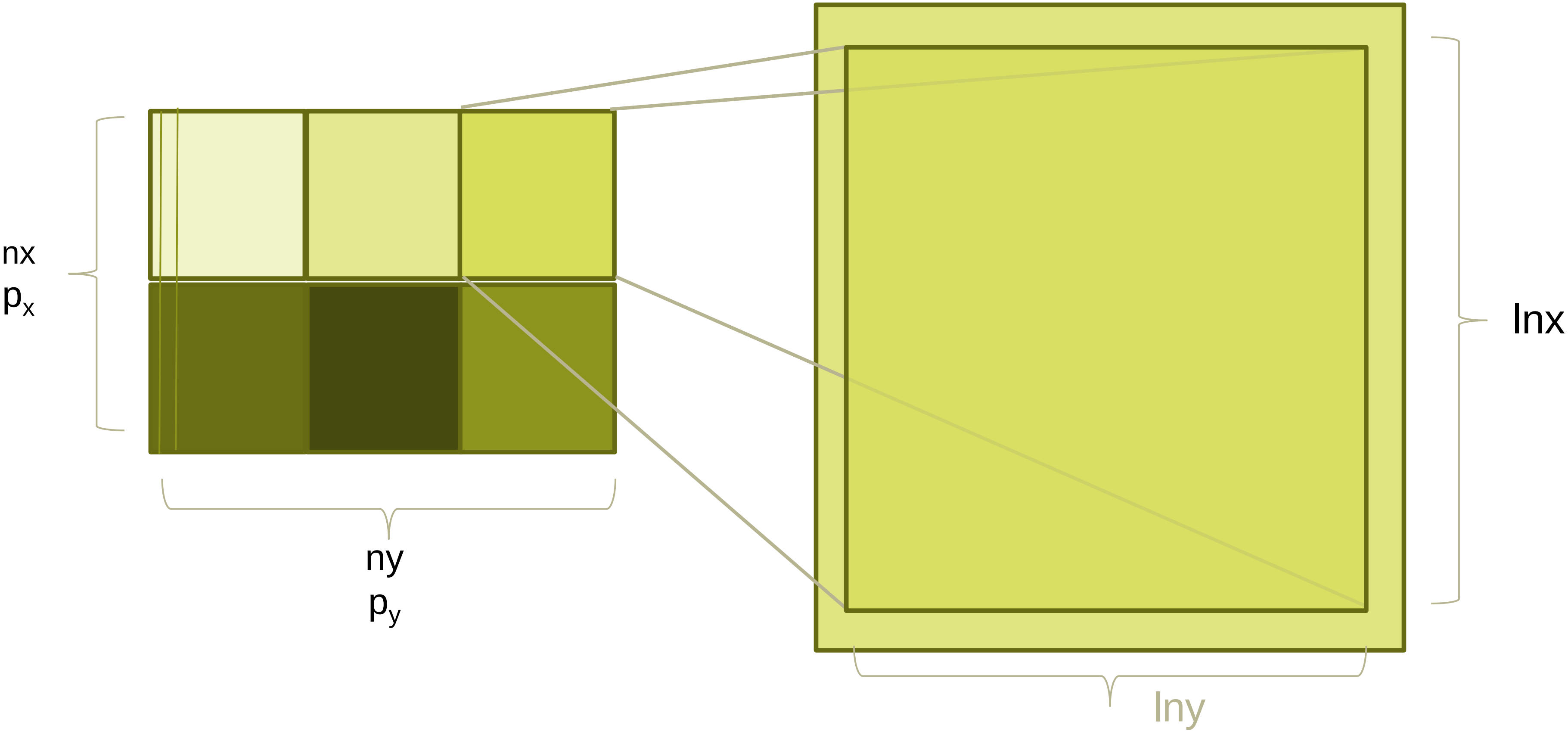


MPI-IO Example

Storing a distributed Domain into a single File

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 :



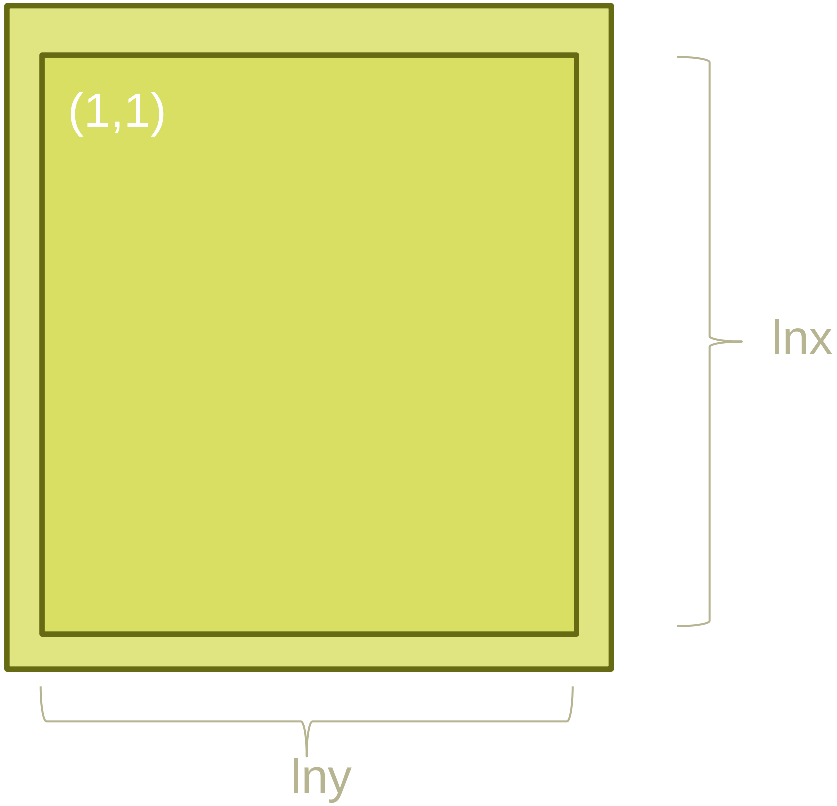
Approach for writing the file

- 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
- Finally we do the I/O

Basic MPI setup

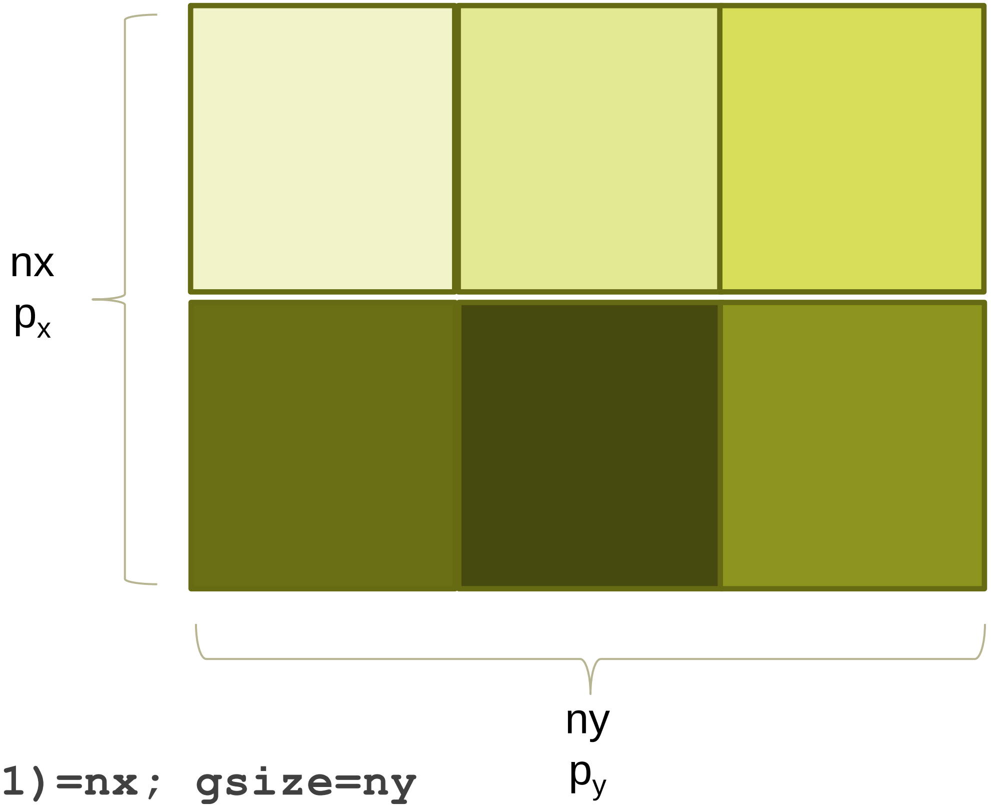
```
nx=512; ny=512 ! Global Domain Size
call MPI_Init(mpierr)
call MPI_Comm_size(MPI_COMM_WORLD, mysize, mpierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, mpierr)

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

References

- **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/**