Optimizing I/O on the Cray XE/XC

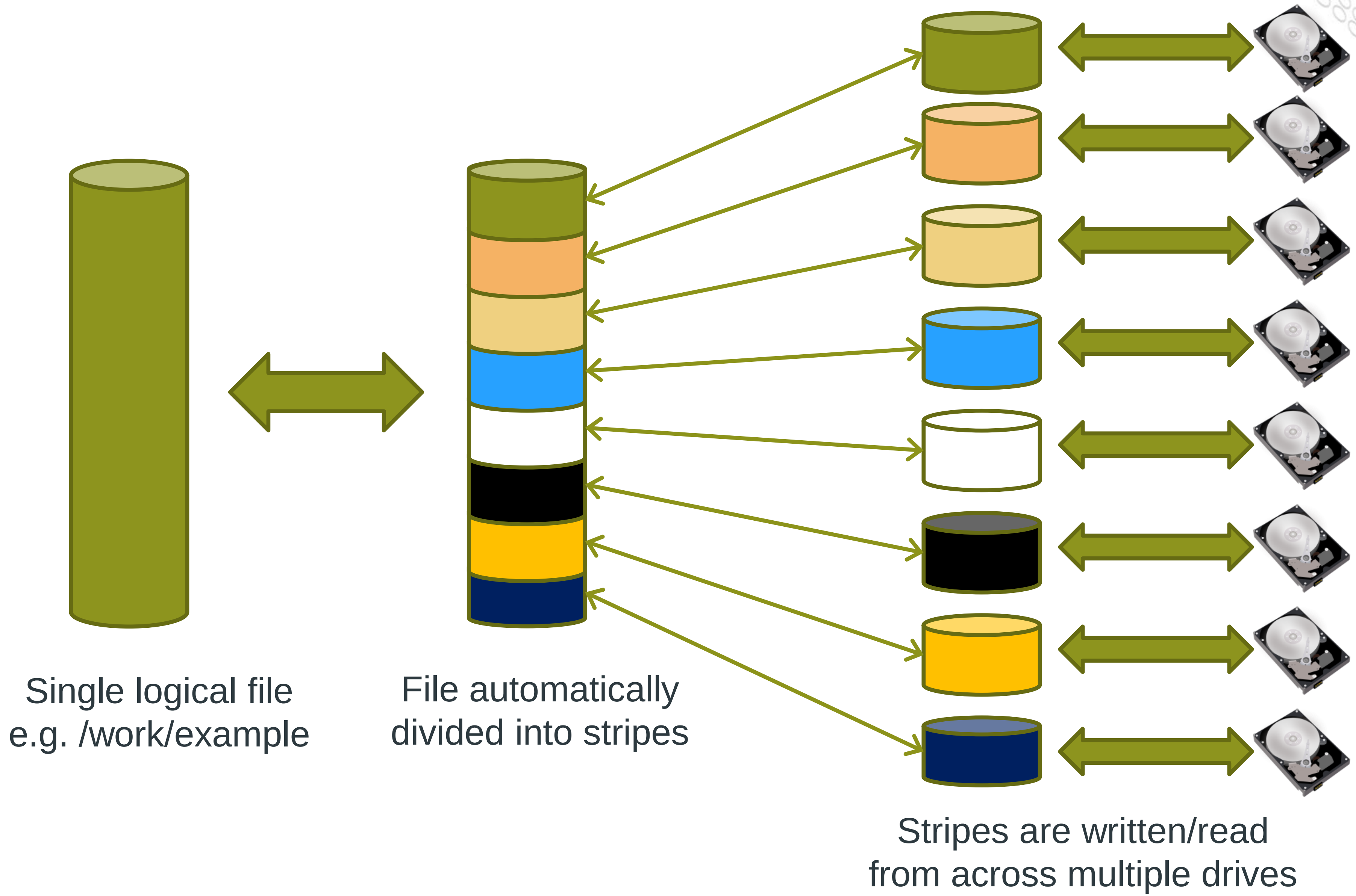
Agenda

- Scaling I/O
- Parallel Filesystems and Lustre
- I/O patterns
- Optimising I/O

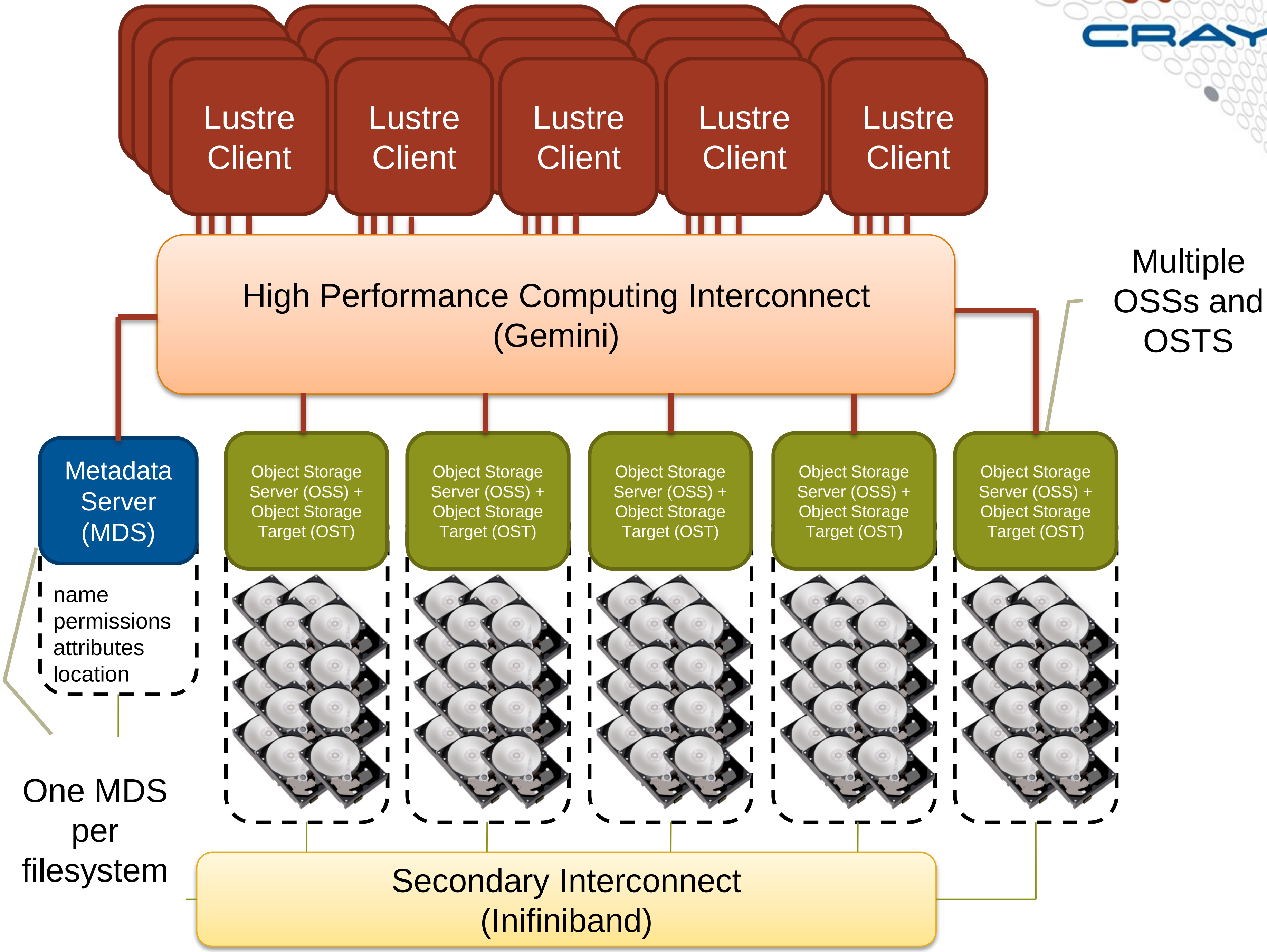
Scaling I/O

- **We tend to talk a lot about scaling application computation**
  - Nodes, network, software
- **We need I/O to scale**
- **So let us now consider filesystems and storage…**

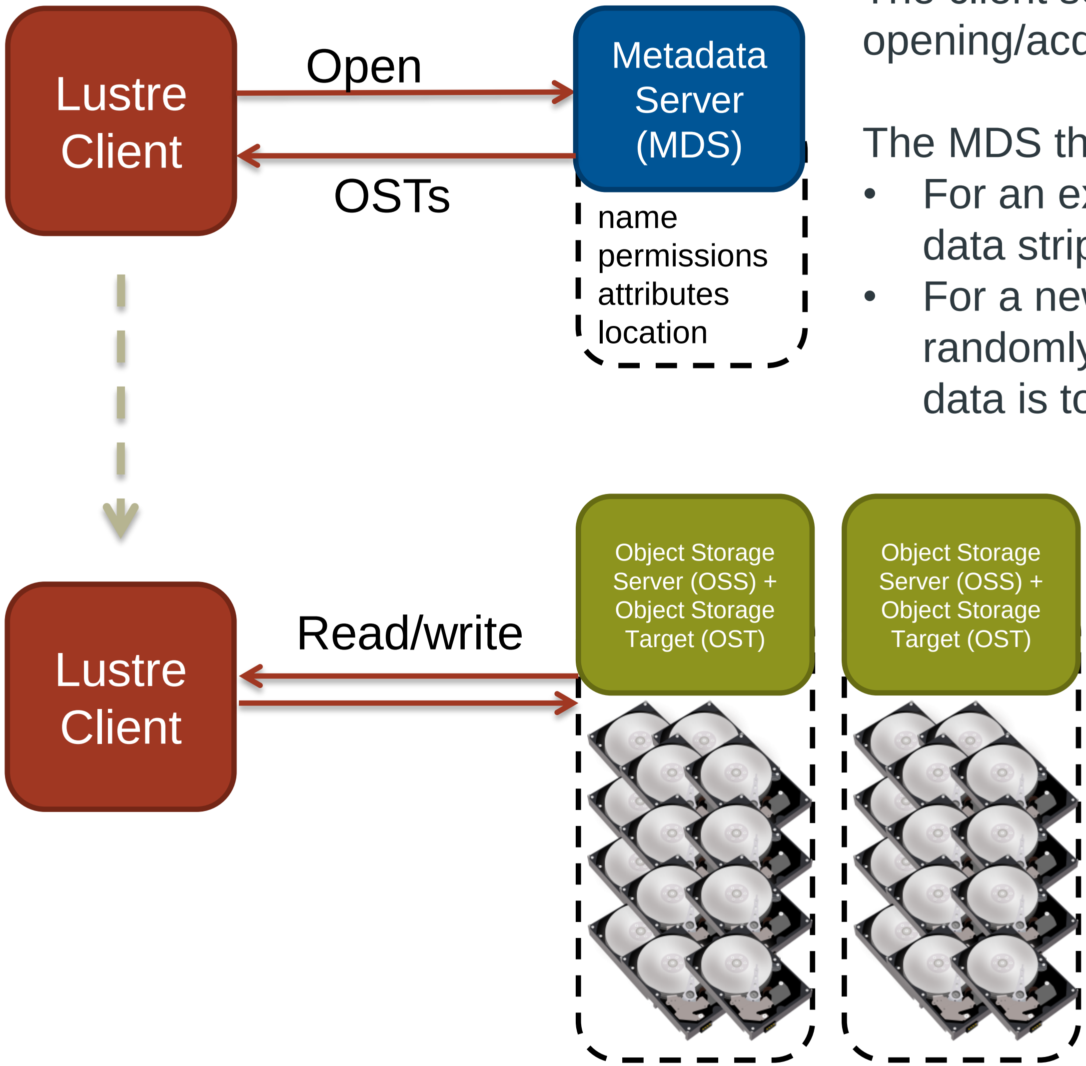
Parallel Filesystem fundamentals



- **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)
Multiple OSSs and OSTS
Metadata Server (MDS)
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)
name
permissions
attributes
location
One MDS per filesystem
Secondary Interconnect (Inifiniband)

Opening a file



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

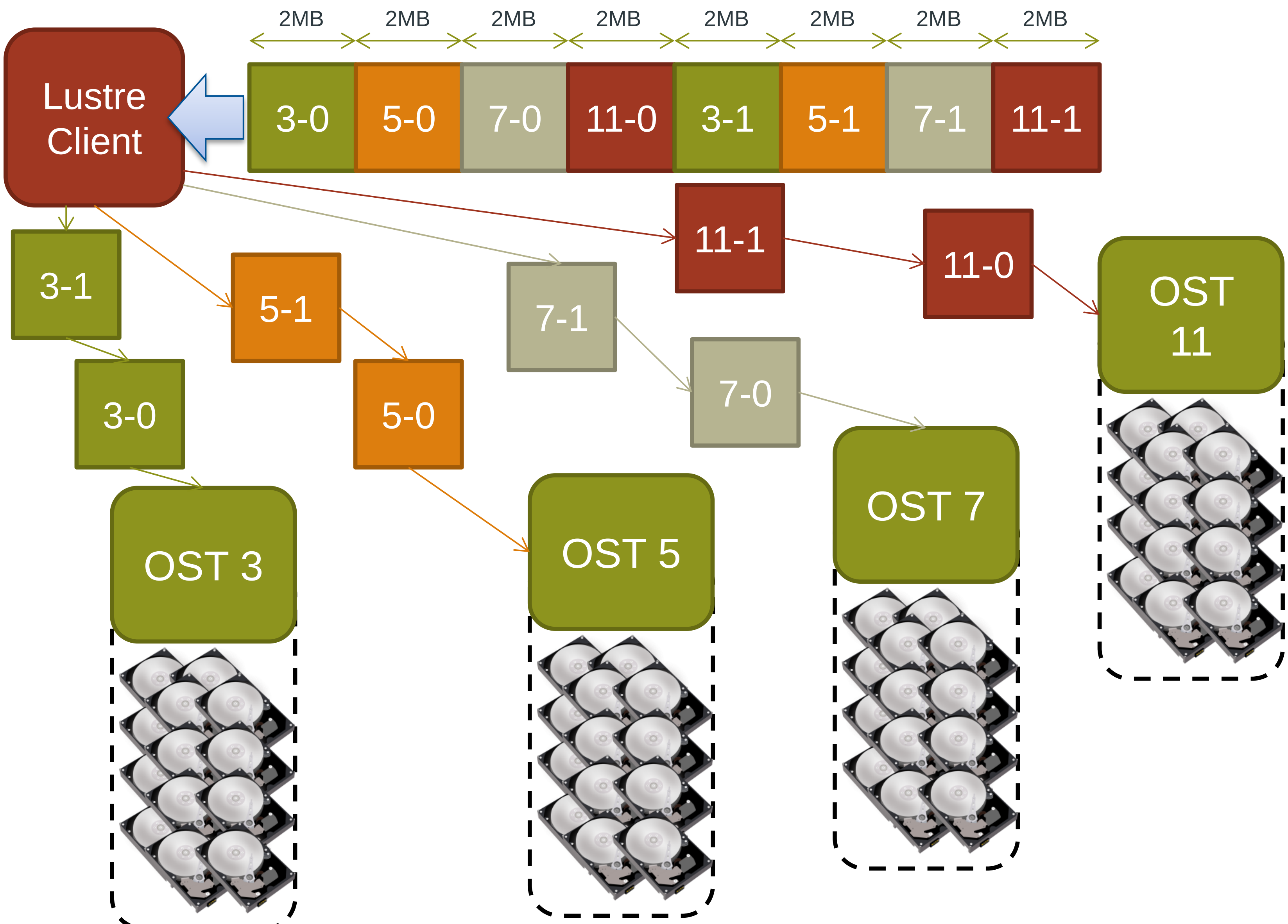
The MDS then passes back a list of OSTs

- 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

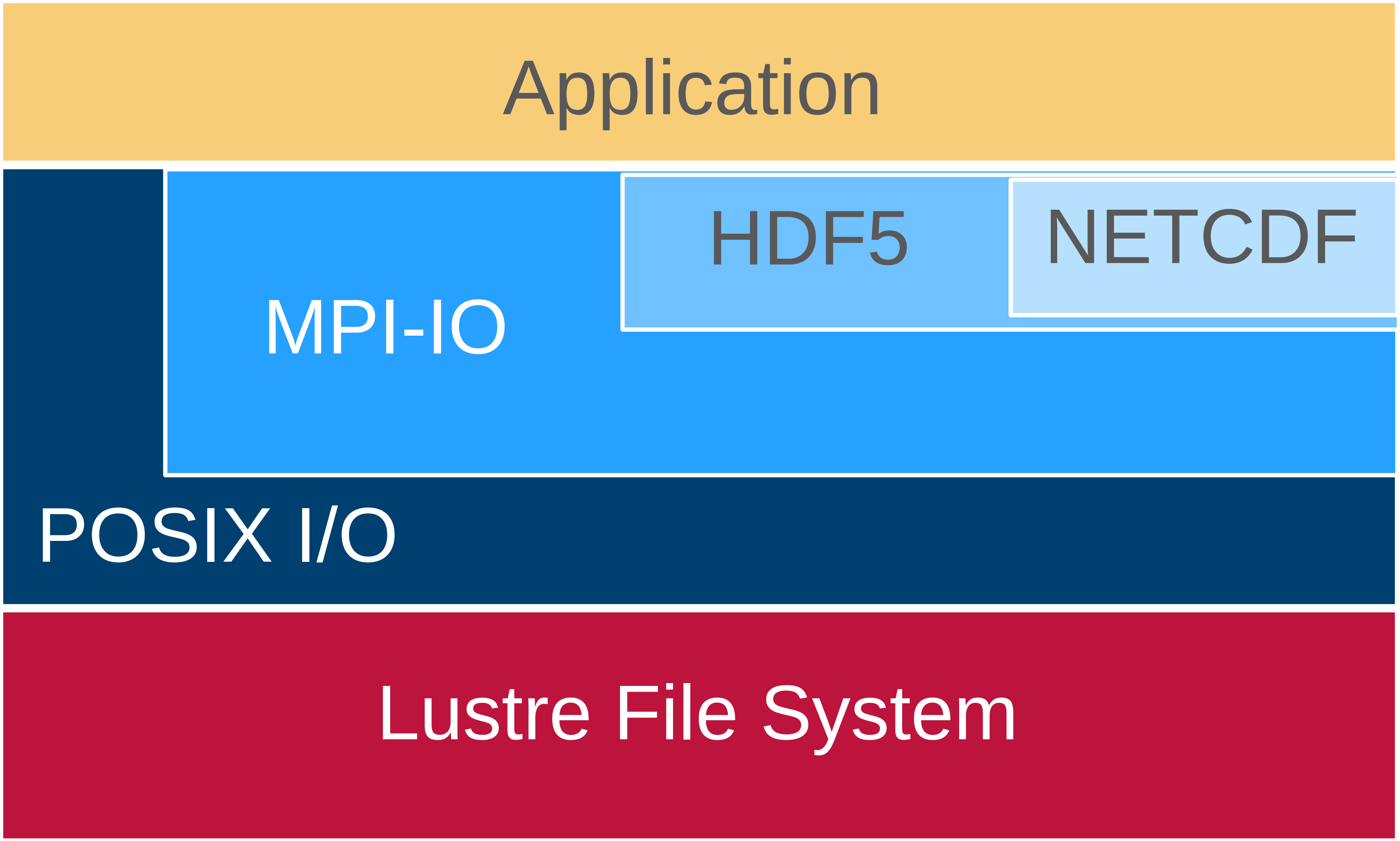
Once a file has been opened no further communication is required between the client and the MDS

All transfer is directly between the assigned OSTs and the client

File decomposition – 2 Megabyte stripes



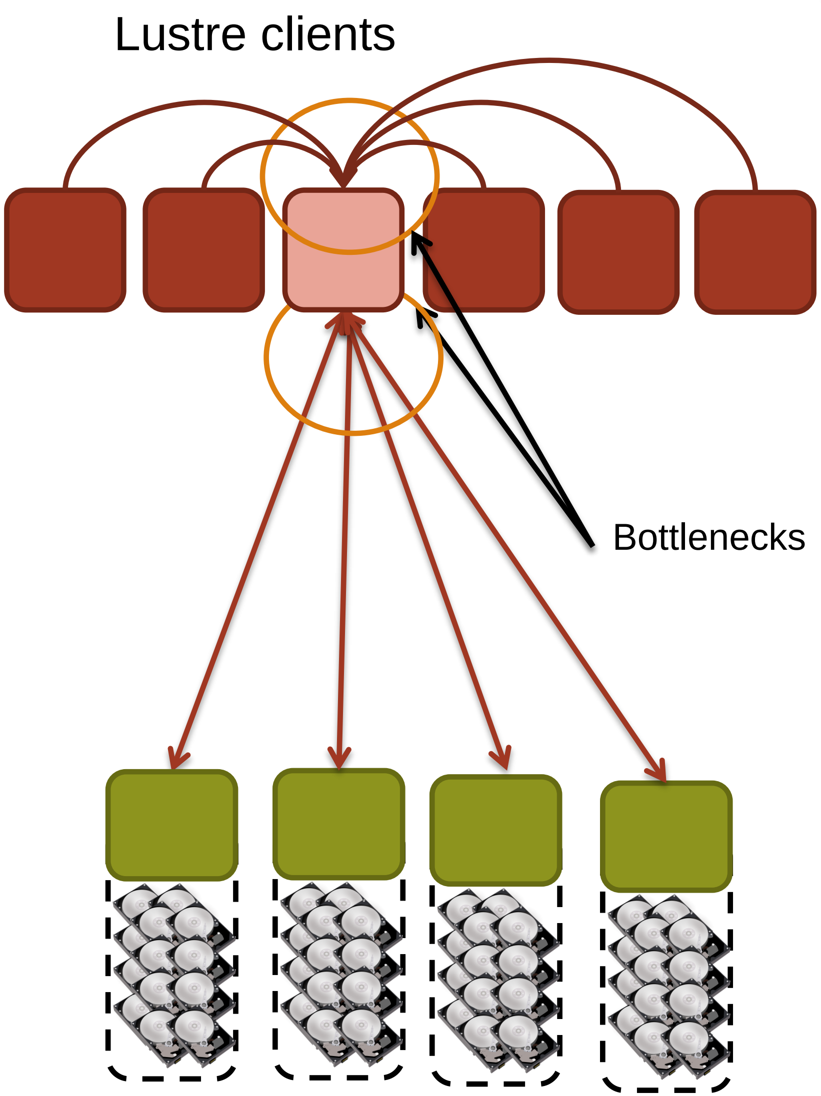
CRAY I/O stack



I/O Patterns

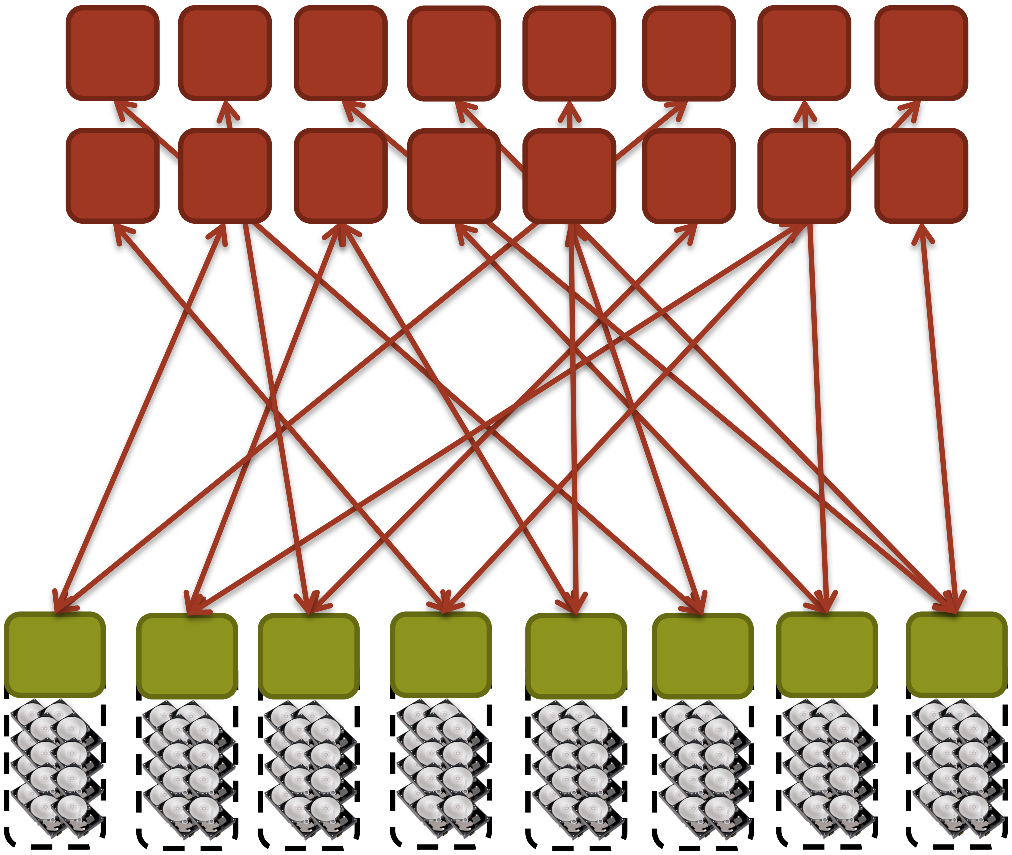
I/O strategies: Spokesperson

- **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 I/O Server**



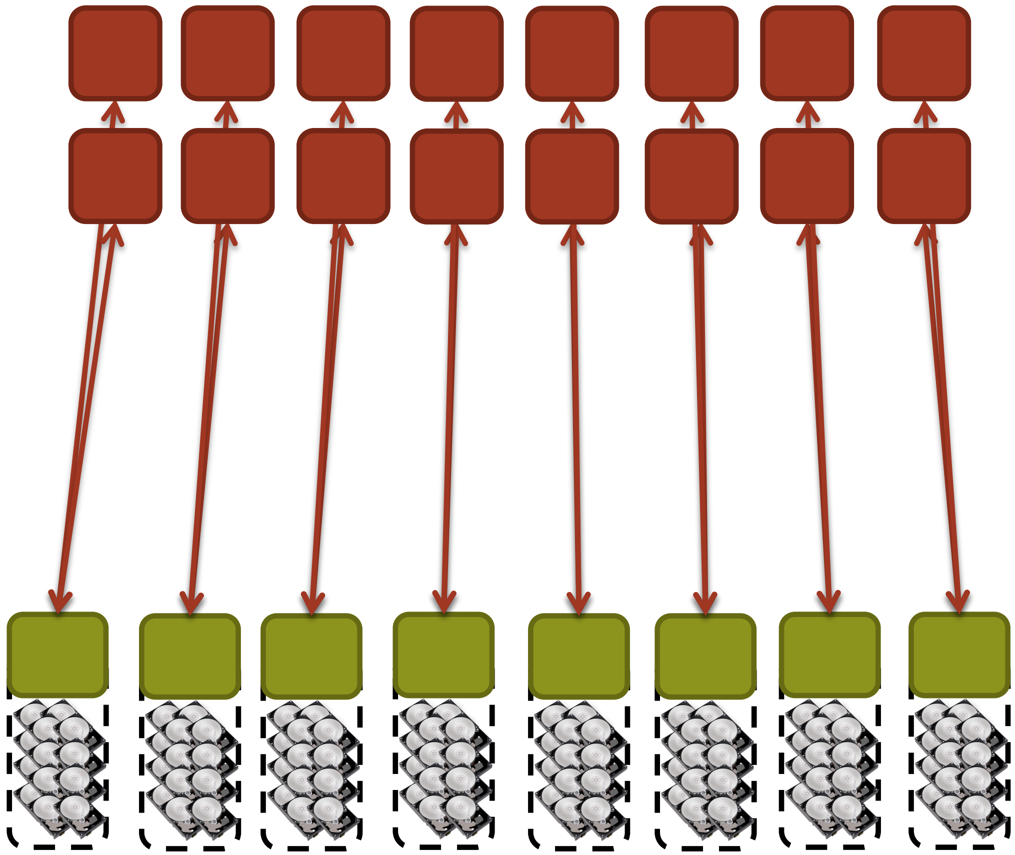
I/O strategies: Multiple Writers – Multiple Files

- **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



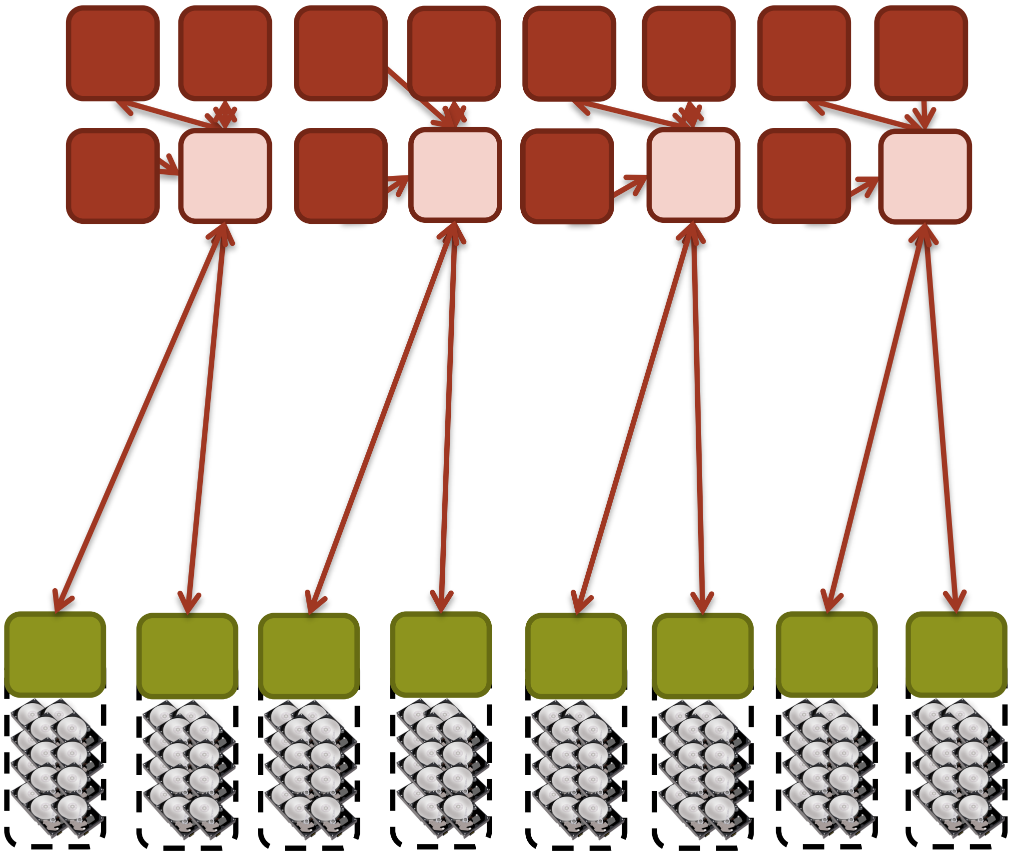
I/O strategies: Multiple Writers – Single File

- **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



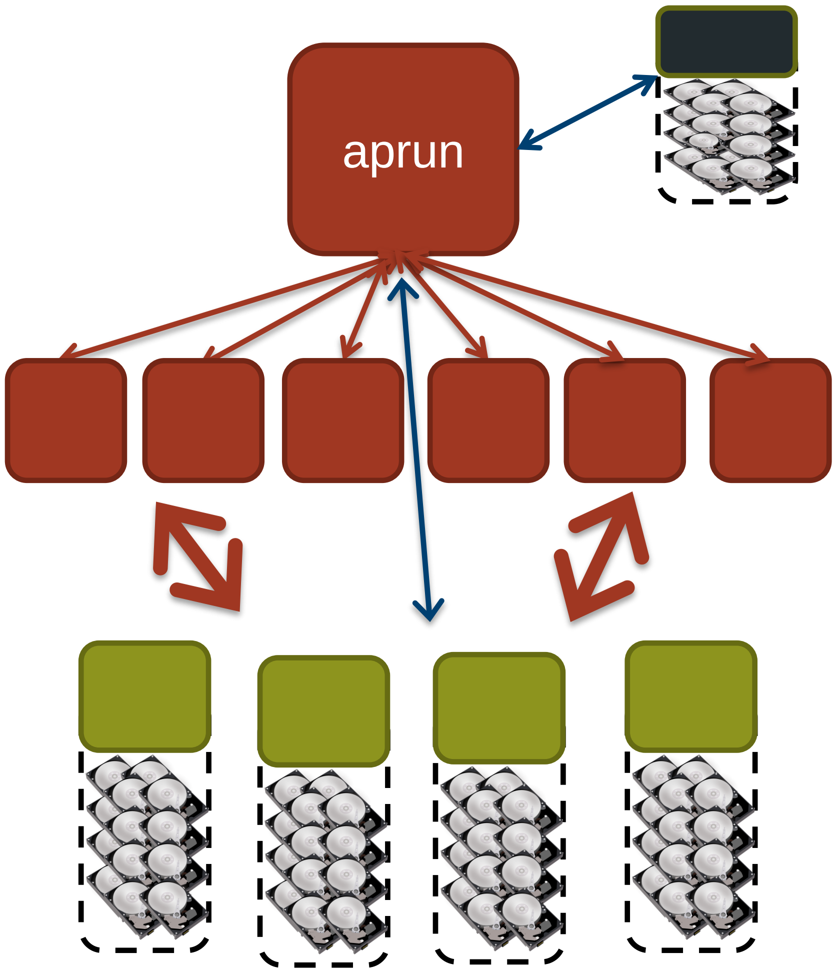
I/O strategies: Collective IO to single or multiple files

- **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

- **All STDIN, STDOUT, and STDERR I/O streams serialize through aprun**
- **Disable debugging messages when running in production mode.**
  - “Hello, I’m task 32,000!”
  - “Task 64,000, made it through loop.”



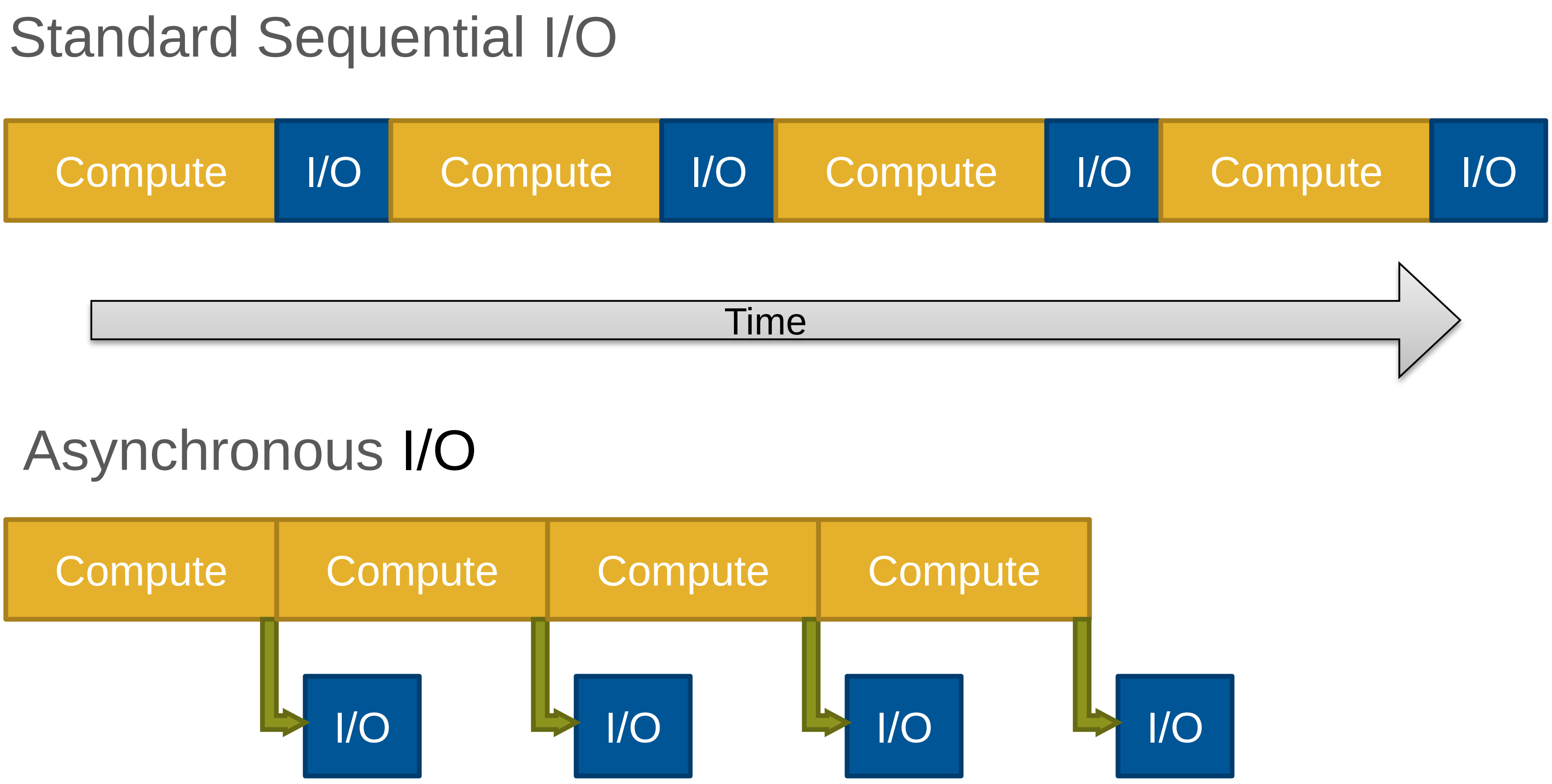
Recipes for good application I/O performance

1. Use Parallel I/O
2. Try to hide I/O (asynchronous I/O)
3. Tune filesystem parameters
4. Use I/O buffering for all sequential I/O

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 filesystem)
- Going to extremes with an I/O pattern will typically lead to problems
- I/O is a shared resource: Expect timing variation

Asynchronous I/O



Asynchronous I/O

- **Majority of data is output**
- **Double buffer arrays to allow computation to continue while data flushed to disk**

1. **Use asynchronous APIs (Fortran, POSIX)**
   - Only covers the I/O call itself, any packing/gathering/encoding still has to be done by the compute processors
   - Extent of achievable overlap depends on library/OS/filesystem
2. **Use 3rd party libraries**
   - Typical examples are MPI I/O
   - 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)

I/O 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)**

Naive I/O Server pseudo code

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
```

Tuning the filesytem: Controlling Lustre striping

- `lfs` **is the Lustre utility for setting the stripe properties of new files, or displaying the striping patterns of existing ones**

- **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.
  - The start index is the only one you can specify, starting with the second OST. In general you have no control over which one is used.

Select best Lustre striping values

- **Selecting the striping values will have a large impact on the I/O performance of your application**
- **Rule of thumb:**
  1. **#files > # OSTs** → Set stripe_count=1
     You will reduce the lustre contention 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!**

Sample Lustre commands: lfs osts

```
crystal:ior% lfs osts
OBDS::
0: snx11014-OST0000_UUID ACTIVE
1: snx11014-OST0001_UUID ACTIVE
2: snx11014-OST0002_UUID ACTIVE
3: snx11014-OST0003_UUID ACTIVE
4: snx11014-OST0004_UUID ACTIVE
5: snx11014-OST0005_UUID ACTIVE
6: snx11014-OST0006_UUID ACTIVE
7: snx11014-OST0007_UUID ACTIVE
8: snx11014-OST0008_UUID ACTIVE
9: snx11014-OST0009_UUID ACTIVE
10: snx11014-OST000a_UUID ACTIVE
11: snx11014-OST000b_UUID ACTIVE
12: snx11014-OST000c_UUID ACTIVE
13: snx11014-OST000d_UUID ACTIVE
14: snx11014-OST000e_UUID ACTIVE
15: snx11014-OST000f_UUID ACTIVE
16: snx11014-OST0010_UUID ACTIVE
…
```

Sample Lustre commands: lfs df

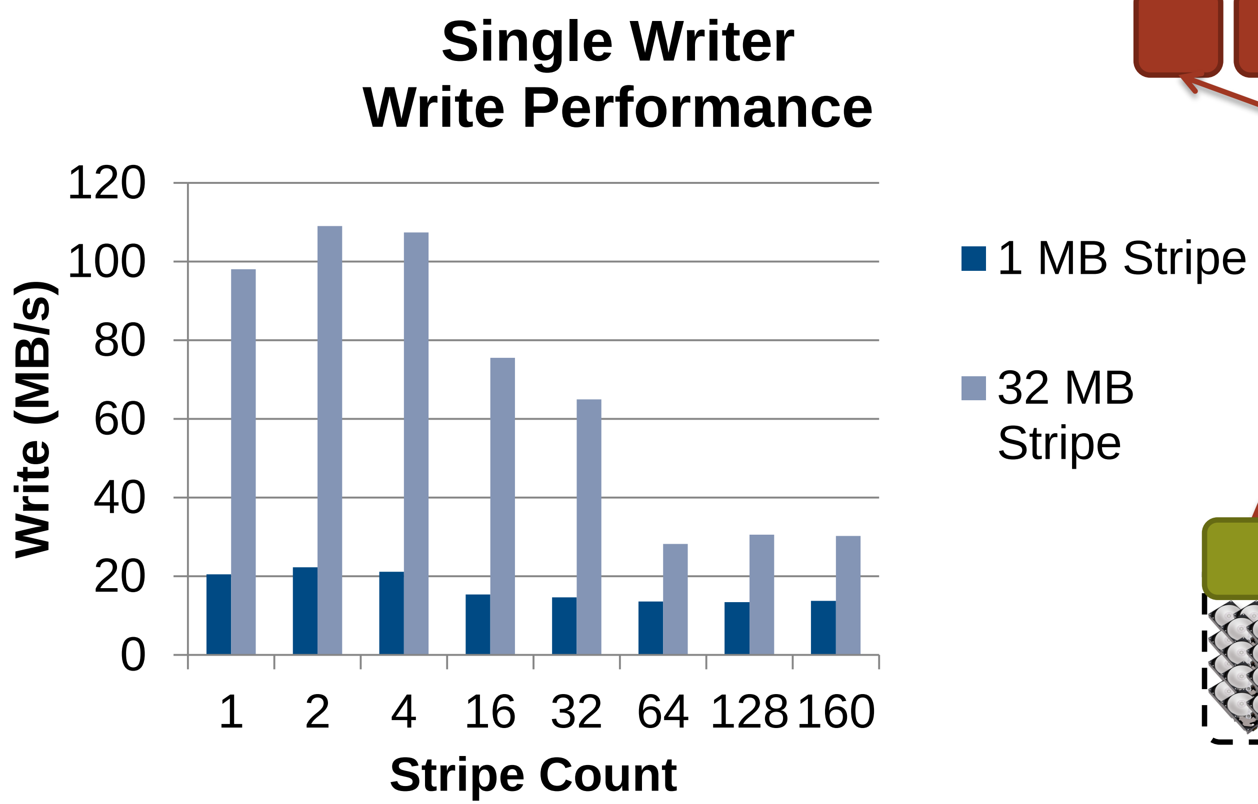
```
crystal:ior% lfs df -h
UUID                          bytes        Used   Available Use% Mounted on
snx11014-MDT0000_UUID          2.1T       47.5G        2.0T   2% /lus/sonexion[MDT:0]
snx11014-OST0000_UUID         20.8T        4.6T       16.0T  22% /lus/sonexion[OST:0]
snx11014-OST0001_UUID         20.8T        4.3T       16.3T  21% /lus/sonexion[OST:1]
snx11014-OST0002_UUID         20.8T        4.3T       16.3T  21% /lus/sonexion[OST:2]
snx11014-OST0003_UUID         20.8T        4.0T       16.6T  20% /lus/sonexion[OST:3]
snx11014-OST0004_UUID         20.8T        4.3T       16.3T  21% /lus/sonexion[OST:4]
snx11014-OST0005_UUID         20.8T        4.6T       16.0T  22% /lus/sonexion[OST:5]
snx11014-OST0006_UUID         20.8T        3.9T       16.7T  19% /lus/sonexion[OST:6]
snx11014-OST0007_UUID         20.8T        4.0T       16.6T  20% /lus/sonexion[OST:7]
snx11014-OST0008_UUID         20.8T        4.4T       16.2T  22% /lus/sonexion[OST:8]
snx11014-OST0009_UUID         20.8T        5.1T       15.5T  25% /lus/sonexion[OST:9]
snx11014-OST000a_UUID         20.8T        4.9T       15.8T  24% /lus/sonexion[OST:10]
snx11014-OST000b_UUID         20.8T        4.5T       16.2T  22% /lus/sonexion[OST:11]
snx11014-OST000c_UUID         20.8T        4.8T       15.8T  23% /lus/sonexion[OST:12]
…
snx11014-OST001d_UUID         20.8T        4.1T       16.5T  20% /lus/sonexion[OST:29]
snx11014-OST001e_UUID         20.8T        3.6T       17.0T  18% /lus/sonexion[OST:30]
snx11014-OST001f_UUID         20.8T        3.6T       17.0T  18% /lus/sonexion[OST:31]

filesystem summary:          666.9T      137.2T      522.9T  21% /lus/sonexion
```

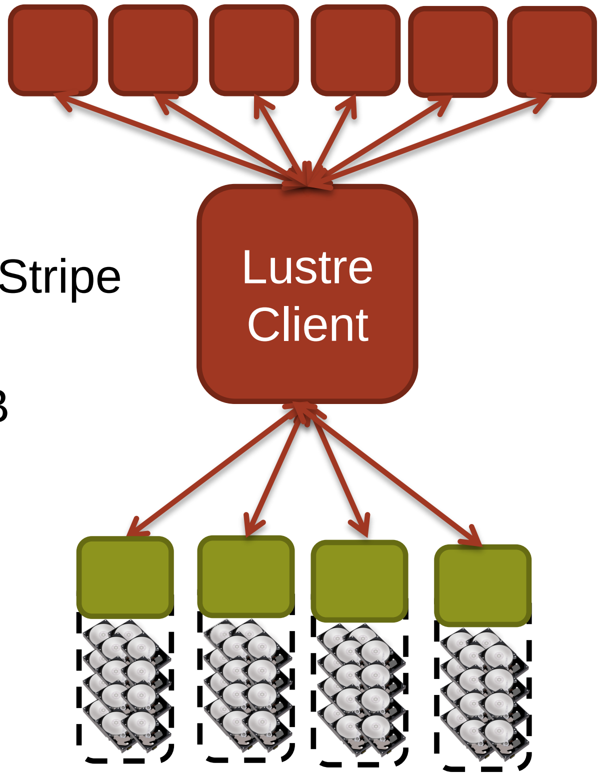
Sample Lustre commands: striping

```
crystal:ior% mkdir tigger
crystal:ior% lfs setstripe -s 2m -c 4 tigger
crystal:ior% lfs getstripe tigger
tigger
stripe_count:   4 stripe_size:    2097152 stripe_offset:  -1
crystal% cd tigger
crystal:tigger% ~/tools/mkfile_linux/mkfile 2g 2g
crystal:tigger% ls -lh 2g
-rw------T 1 harveyr criemp 2.0G Sep 11 07:50 2g
crystal:tigger% lfs getstripe 2g
2g
lmm_stripe_count:   4
lmm_stripe_size:    2097152
lmm_layout_gen:     0
lmm_stripe_offset:  26
        obdidx           objid          objid            group
            26        33770409      0x2034ba9                0
            10        33709179      0x2025c7b                0
            18        33764129      0x2033321                0
            22        33762112      0x2032b40                0
```

Case Study 1: Spokesman

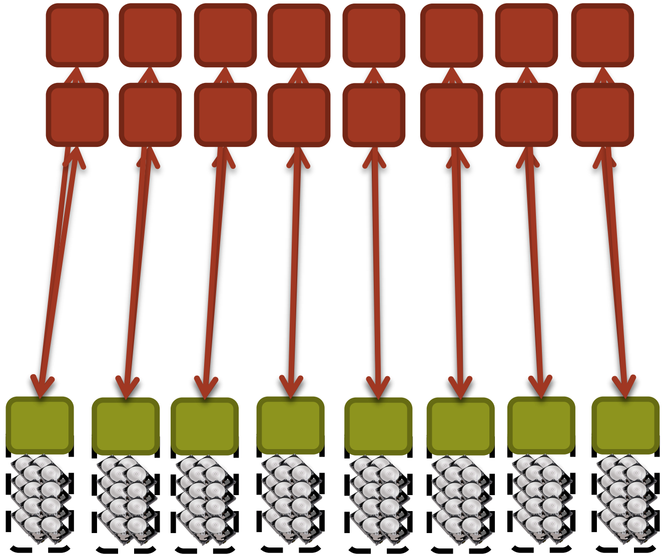
- **32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size**
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance





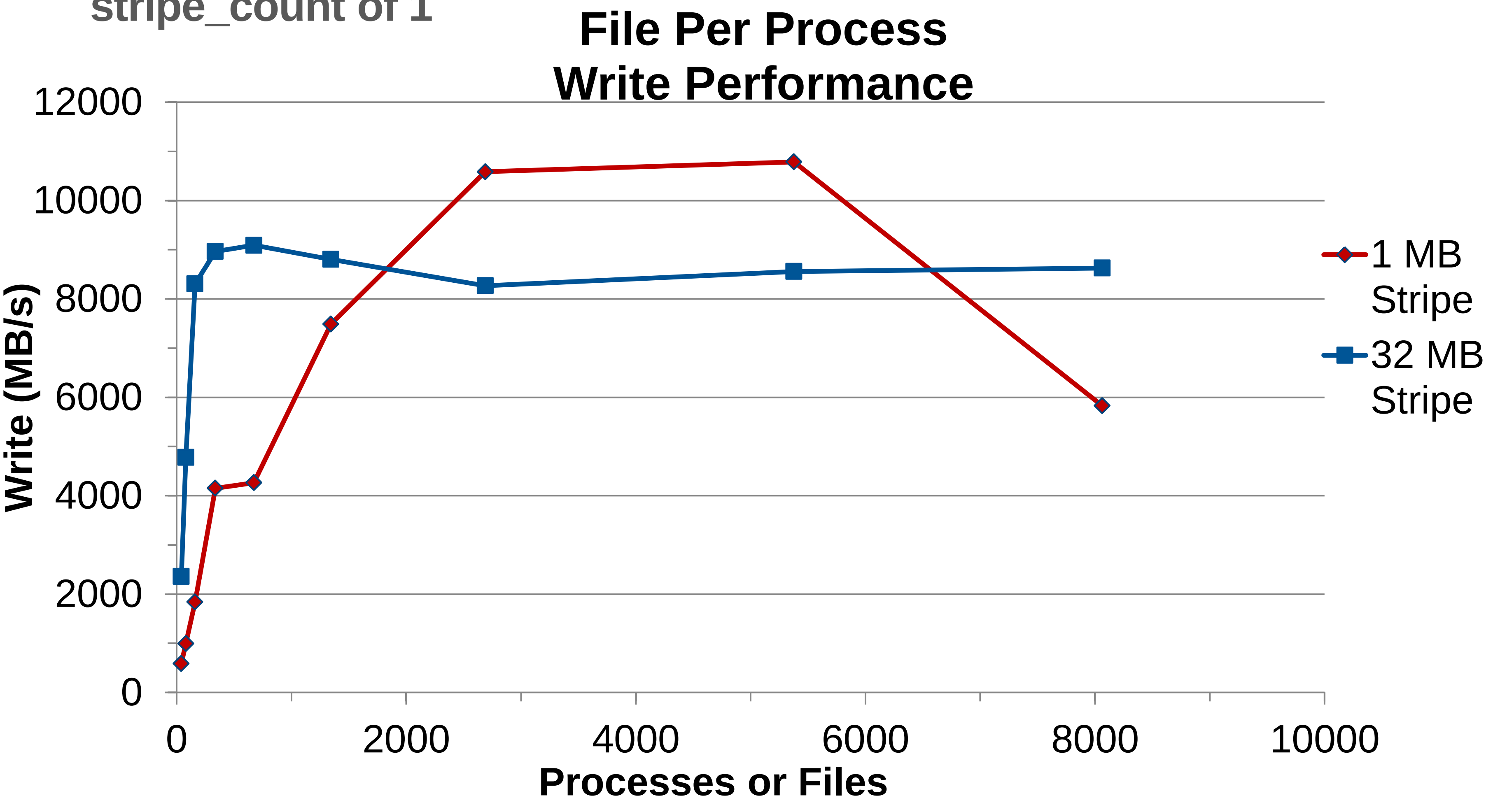
Case Study 2: Parallel I/O into a single file

- **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**

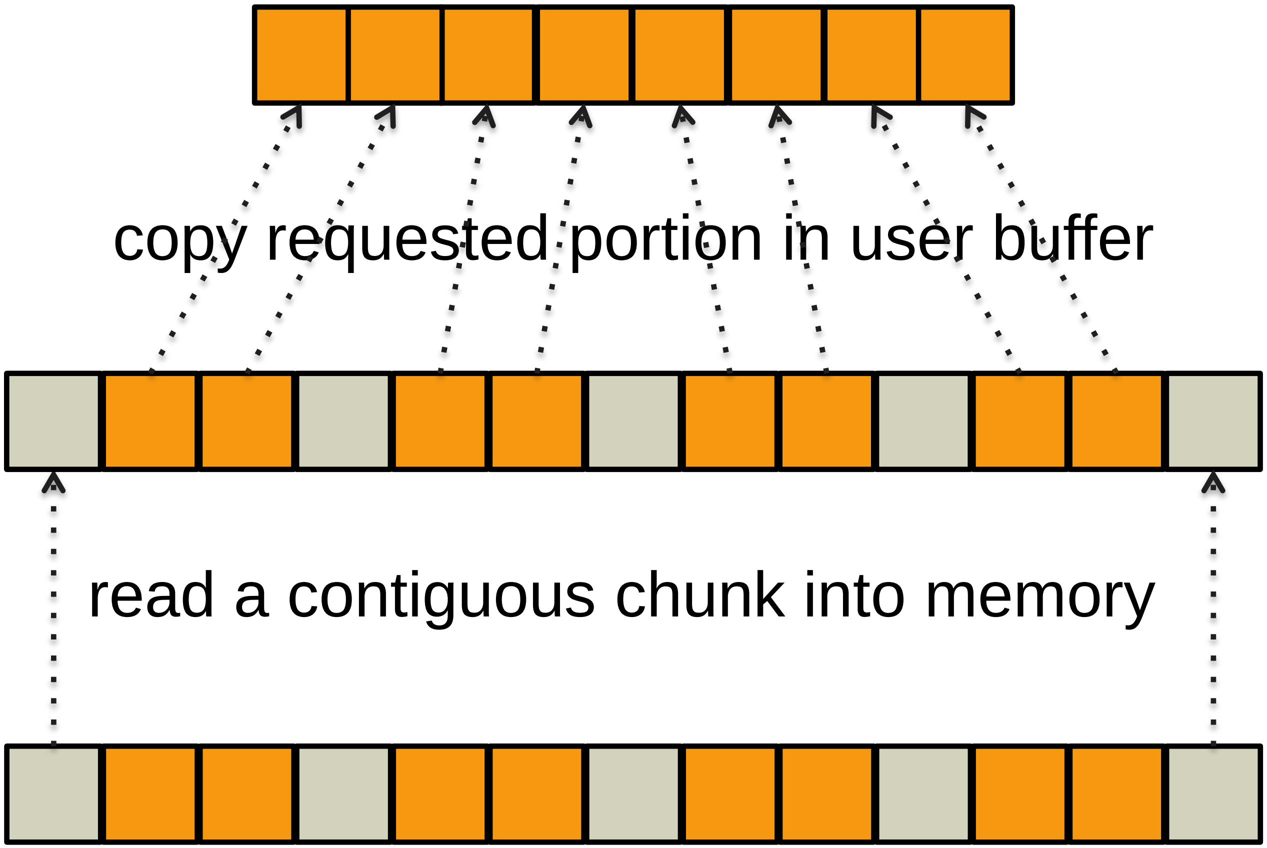


Optimizations and tuning for MPI-I/O

2 Techniques: Sieving and Aggregation

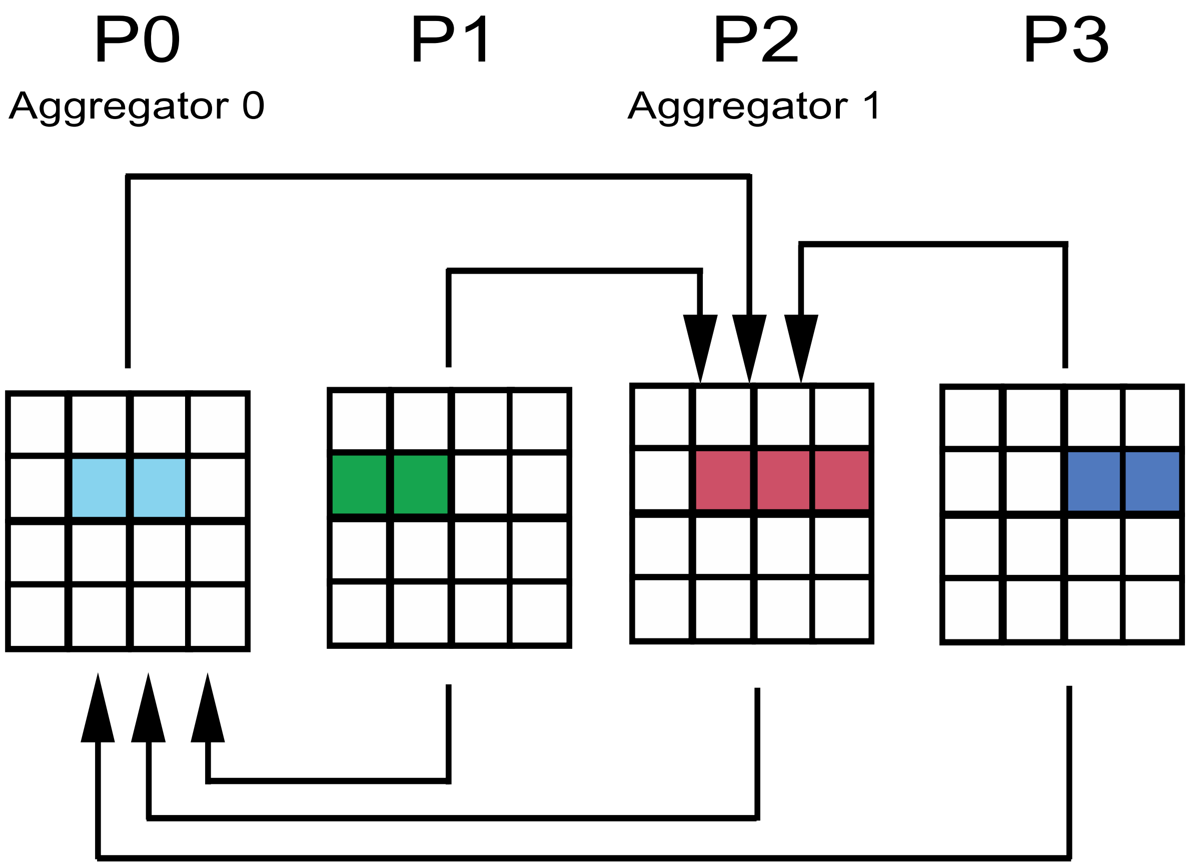
- **Data sieving is used to combine lots of small accesses into a single larger one**
  - Reducing # of operations important (latency)
  - A system buffer/cache is one example
- **Aggregation/Collective Buffering refers to the concept of moving data through intermediate nodes**
  - Different numbers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

Data Sieving

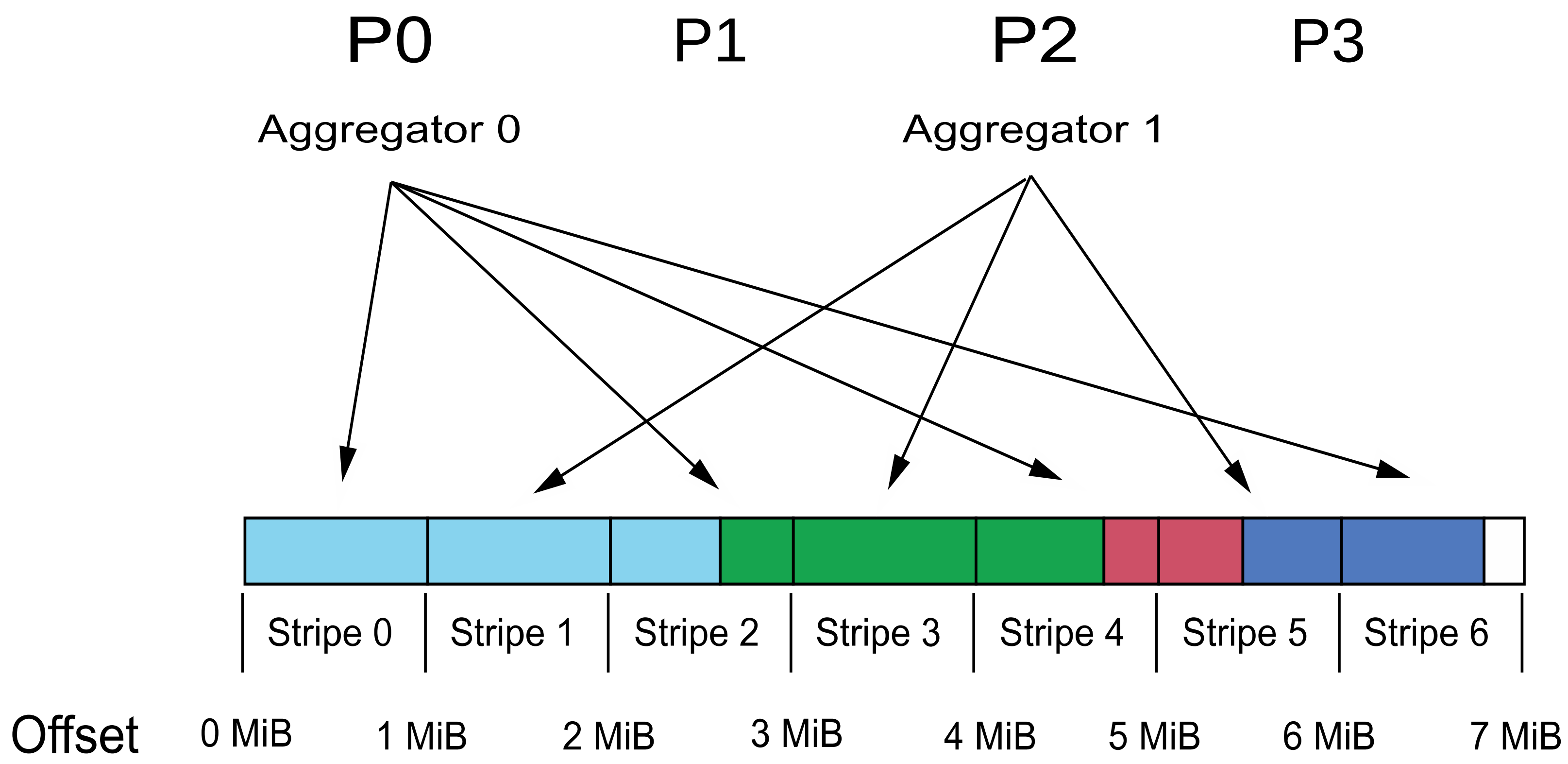


user’s request for non-contiguous data ( ) from a file

Collective buffering: aggregating data

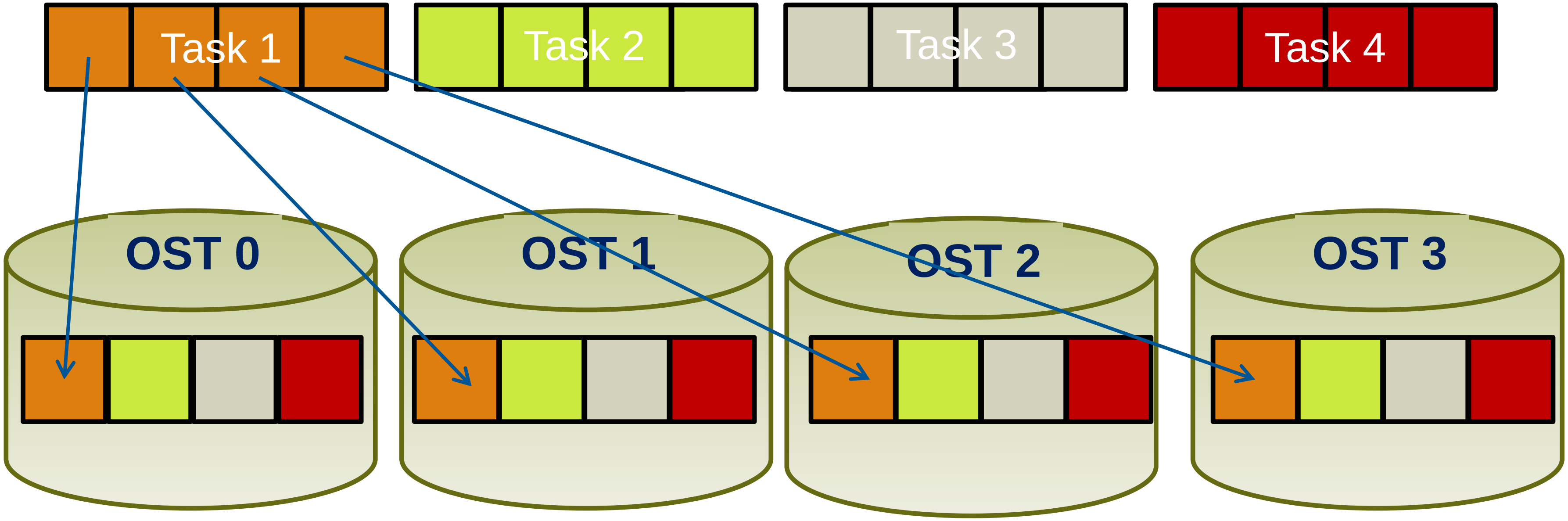


Collective buffering: writing data



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

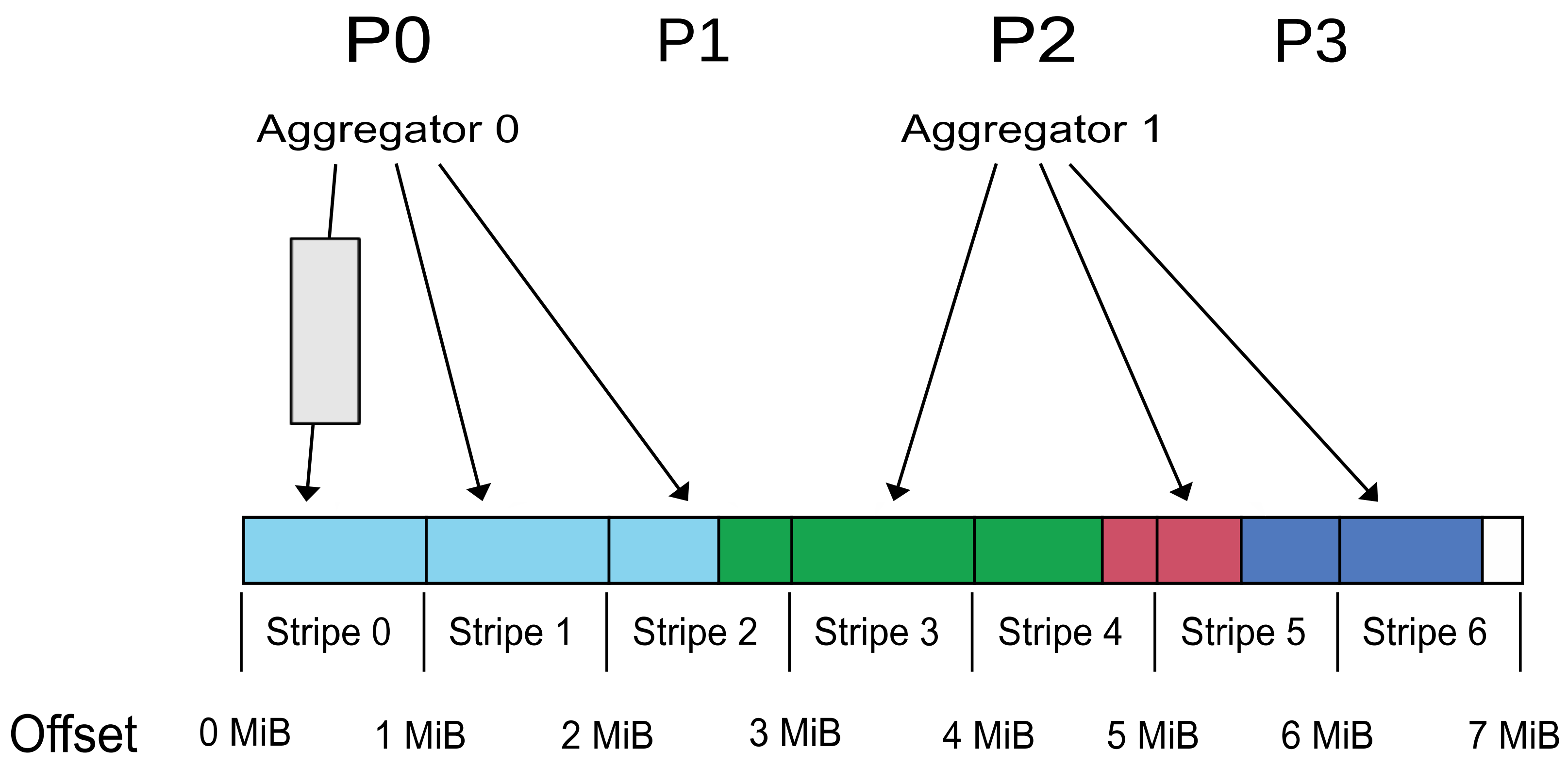
MPI I/O interaction with Lustre

- **Included in the Cray MPT library**
- **Environmental variables used to help MPI-IO optimize I/O performance**
  - MPICH_MPIIO_CB_ALIGN (Default 2) sets collective buffering behavior
  - MPICH_MPIIO_HINTS can set striping_factor and striping_unit for files created with MPI I/O
  - 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 I/O and thus are also affected by these environment variables**

MPICH_MPIIO_CB_ALIGN

- **If set to 2**
  - 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**
  - Not supported (was used for an older algorithm)
- **If set to zero or defined but not assigned a value**
  - Divide the I/O workload equally amongst all aggregators without regard to physical I/O boundaries or Lustre stripes

Collective Buffering writing data CB=2



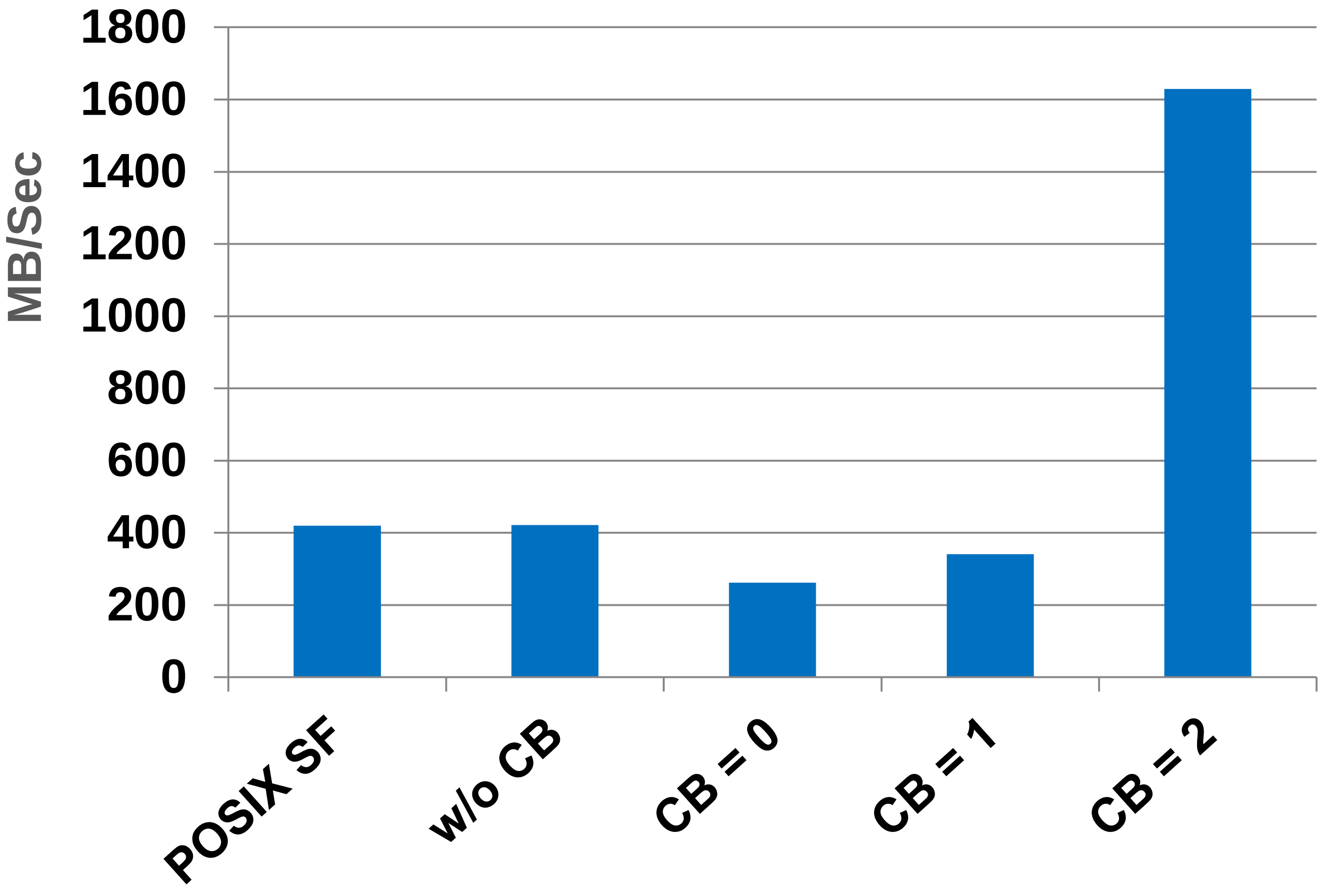
MPI I/O hints (part 1)

- **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

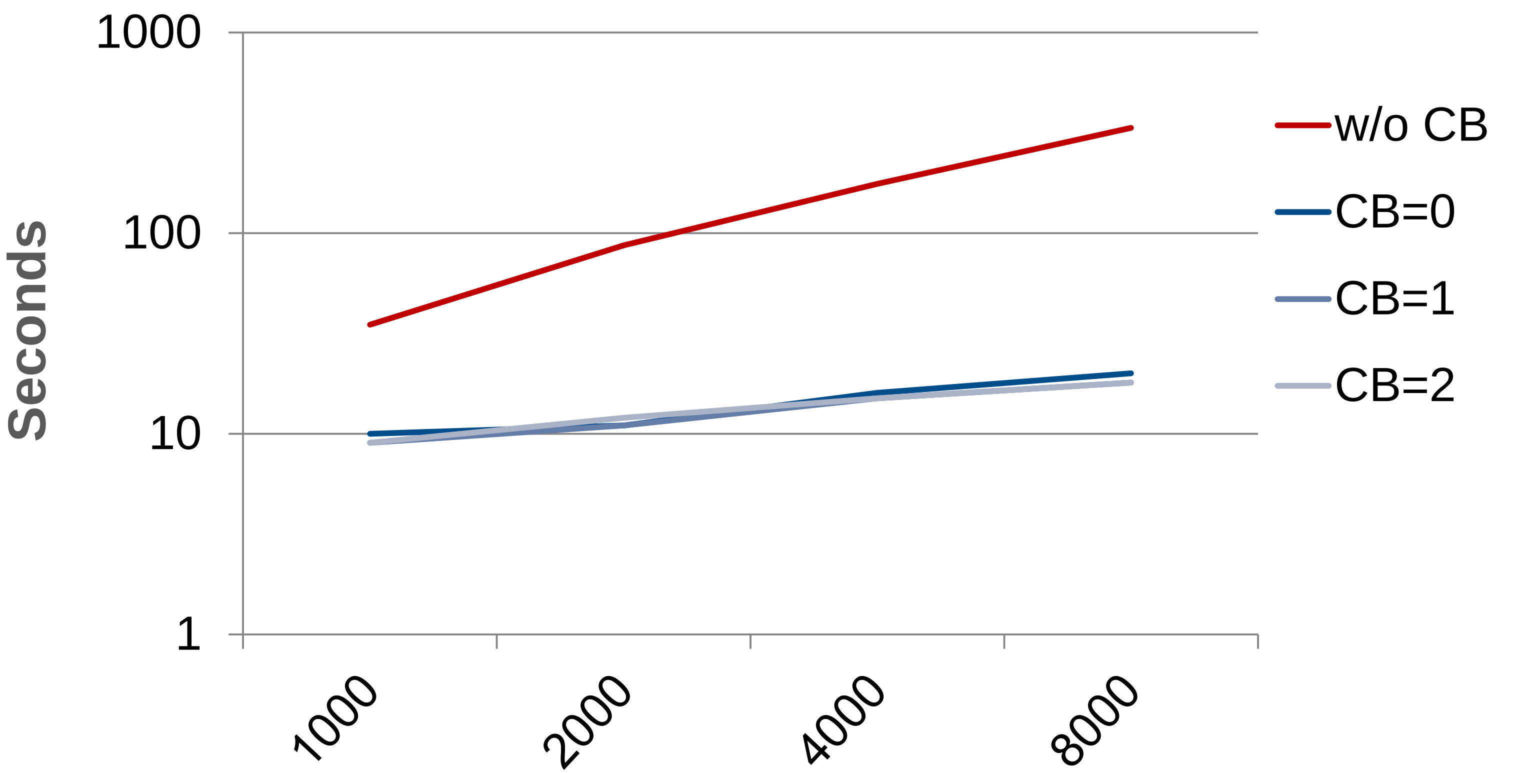
IOR benchmark 1,000,000 bytes

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



HDF5 format dump file from all PEs

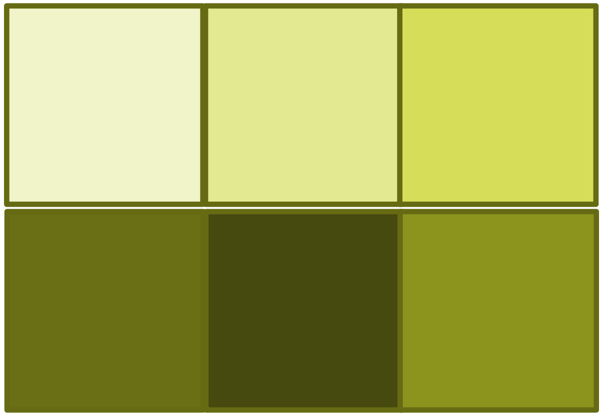
Total file size 6.4 GB. Mesh of 64M bytes 32M elements, with work divided amongst all PEs. Original problem was very poor scaling. For example, without collective buffering, 8000 PEs take over 5 minutes to dump.
Tested on an XT5, 8 stripes, 8 cb_nodes

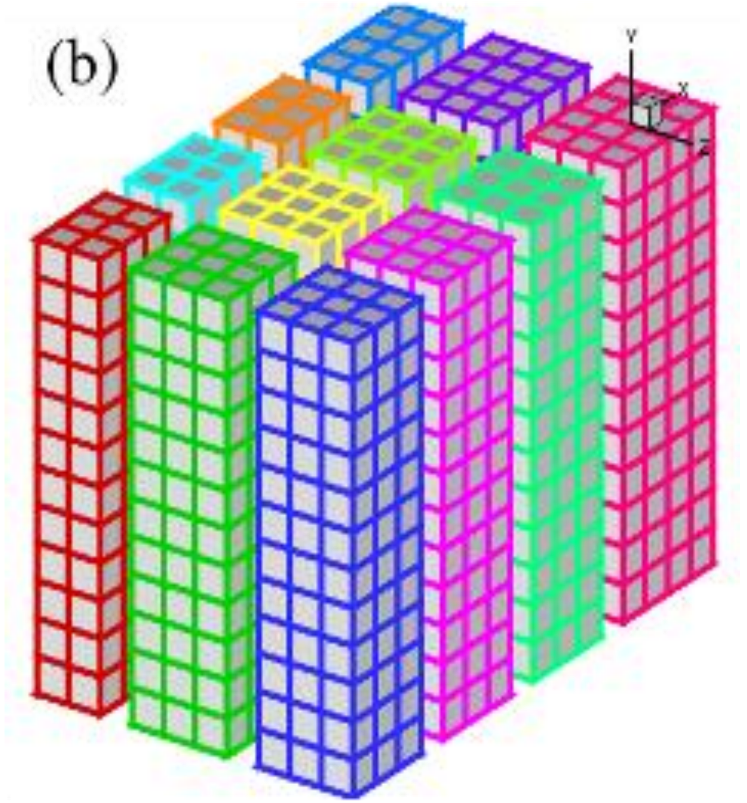


IOBUF

- **IOBUF is a library that intercepts standard I/O (stdio) and enables asynchronous caching and prefetching of sequential file access**
- **Should not be used for**
  - Hybrid programs that do I/O within a parallel region (not thread-safe)
  - Many processes accessing the same file in a coordinated fashion (MPI_File_write/read_all)
- **No need to modify the source code but just**
  - Load the module iobuf
  - Relink your application
  - Set `export IOBUF_PARAMS='*:verbose'` in the batch script
- **See the iobuf(3) manpage**

MPI I/O





- Provides nice features to map data in many processes into one or more files
- In addition you get the performance advantages we talked about so far

Summary

- **I/O is always a bottleneck**
  - Minimize it!
  - You might have to change your I/O implementation when scaling it up
- **Take-home messages on I/O 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
  - Use MPI I/O and/or high-level libraries (HDF5)
- **Lustre rules of thumb**
  - # files > # OSTs => Set stripe_count=1
  - #files==1 => Set stripe_count=#OSTs
  - #files<#OSTs => Select stripe_count so that you use all OSTs

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/**

Backup Slides

MPI-I/O

MPI-I/O

- **Defined by the MPI specification**
- **Allows an application to write into both**
  - distinct files
  - or the same file from multiple MPI processes
- **Uses MPI datatypes to describe both the file and the process data**
- **Supports collective operations**

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

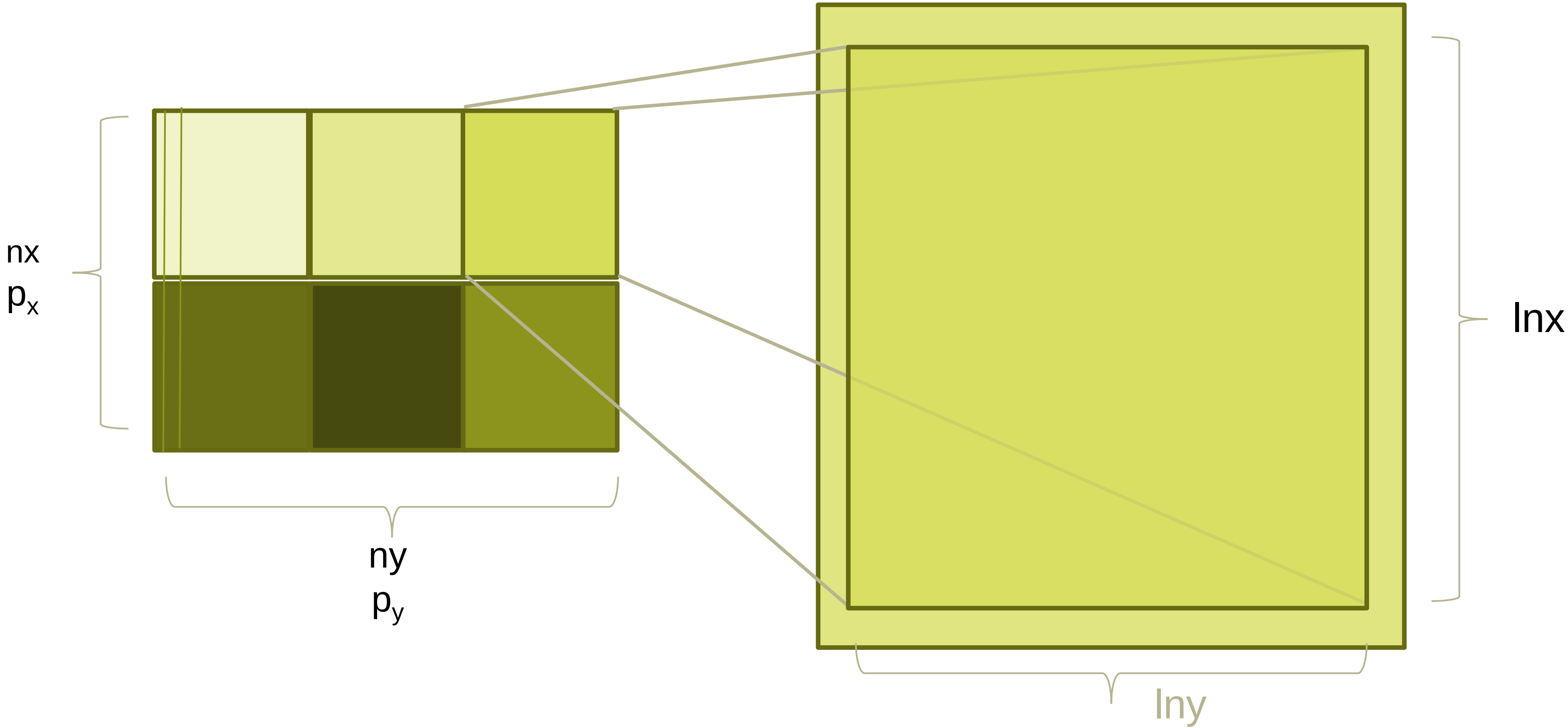
- MPI_File_read_**all**, MPI_File_read_at_**all**, …
- _all indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
- Each process specifies only its own access information – the argument list is the same as for the non-collective functions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

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 :



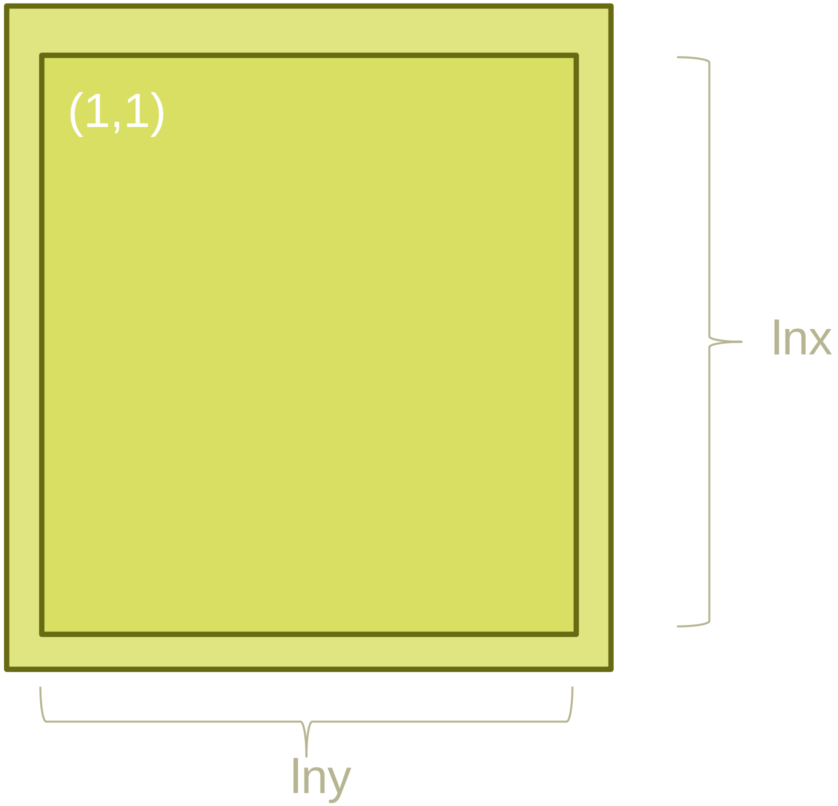
Approach for writing the file

- 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)
```