Optimizing I/O on the Cray XE/XC

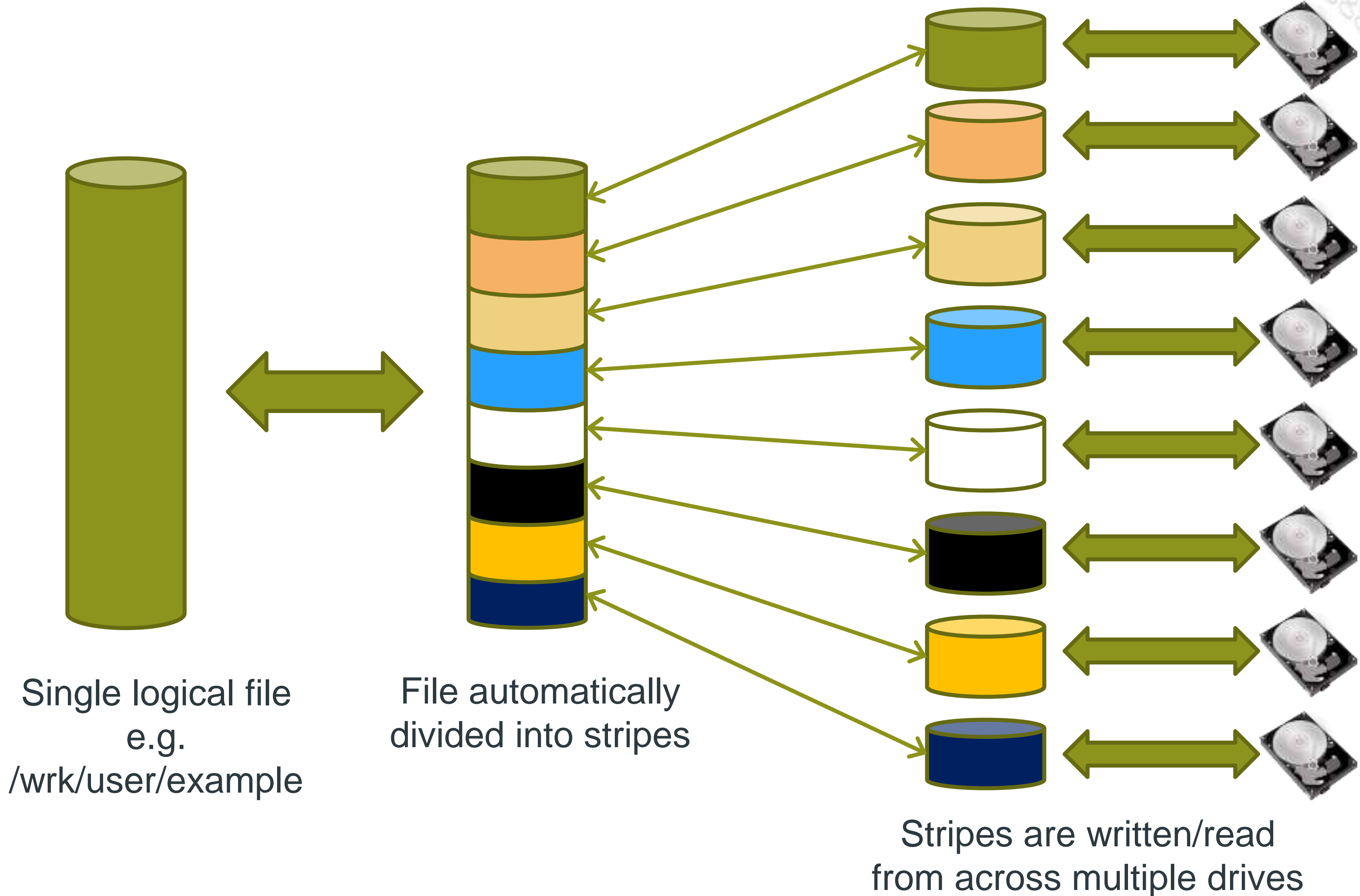
Agenda

- 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…**

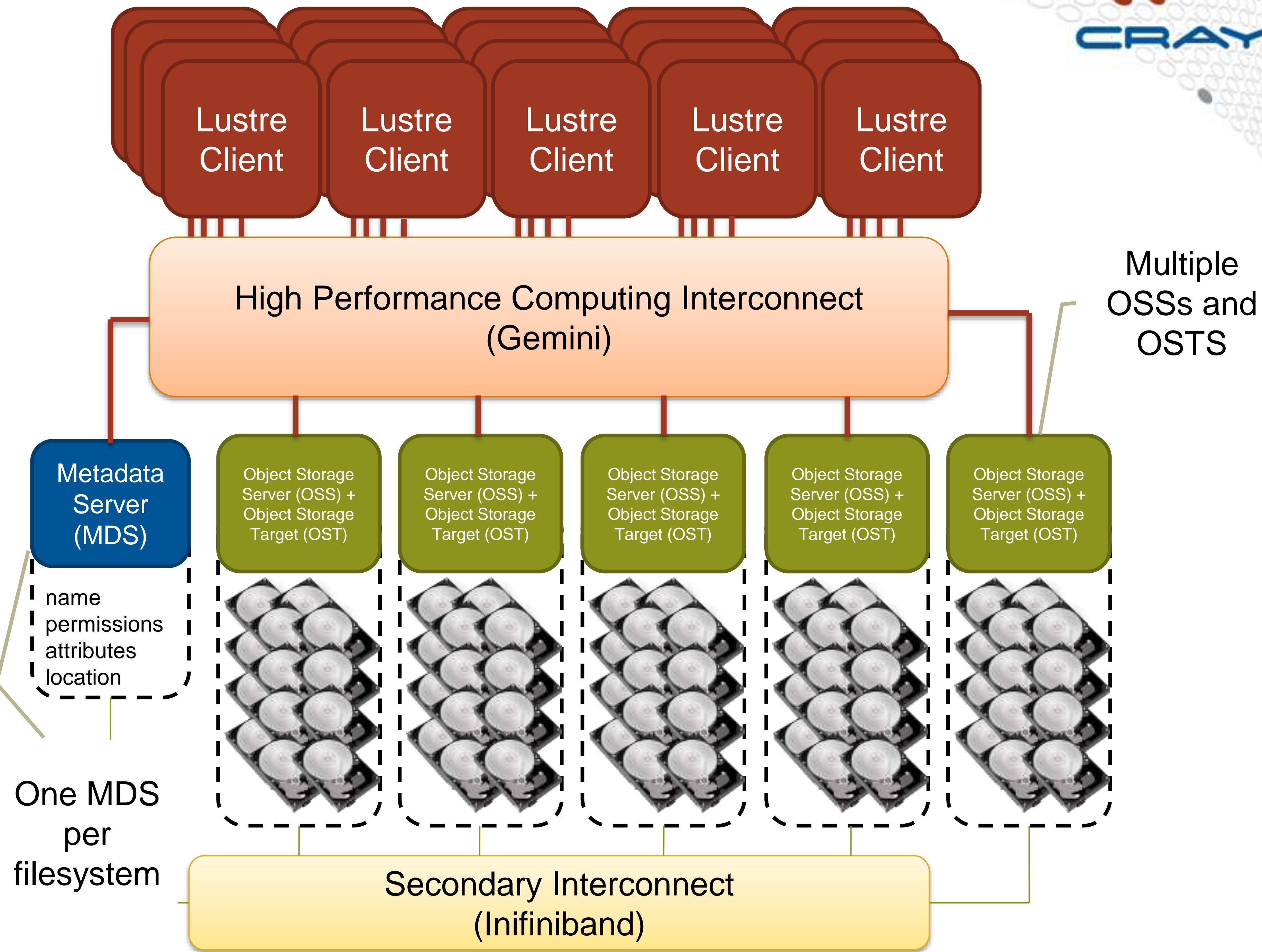
Parallel Filesystem fundamentals



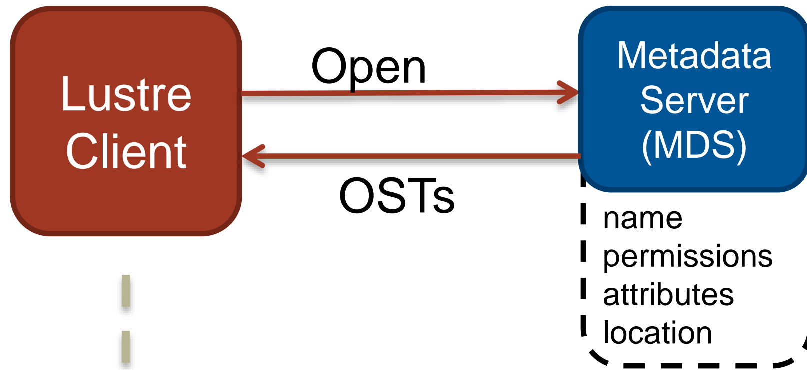
lustre®

CRAY

- **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


CRAY
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)

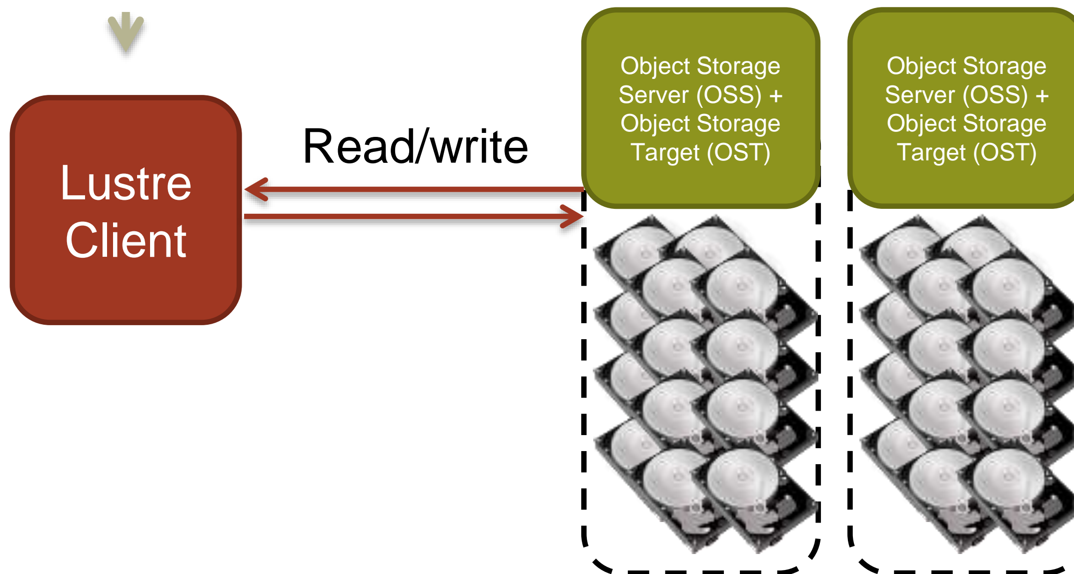
Opening a file



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

The MDS then passes back a list of OSTs

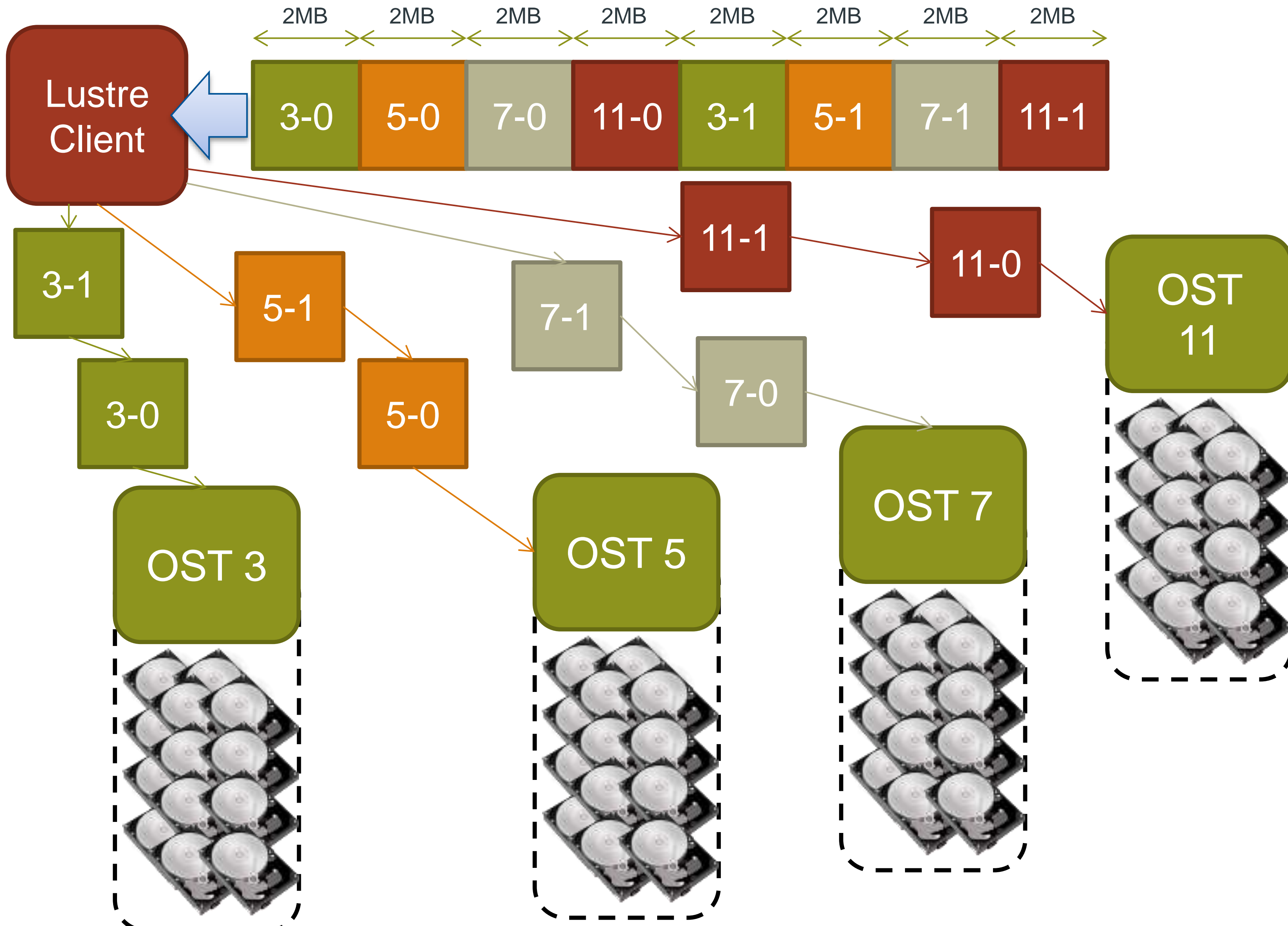
- For an existing file, these contain the data stripes
- For a new files, these typically contain a randomly assigned list of OSTs where data is to be stored



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

Application

HDF5

NETCDF

MPI-IO

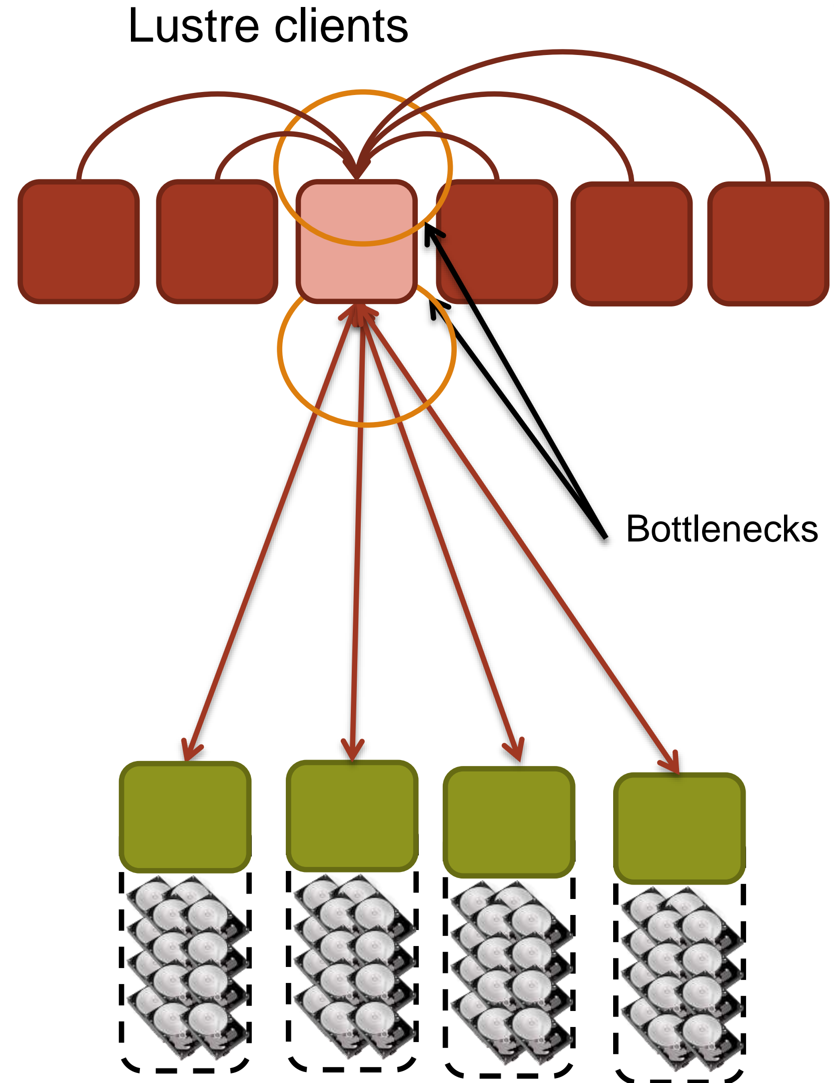
POSIX I/O

Lustre File System

I/O Patterns

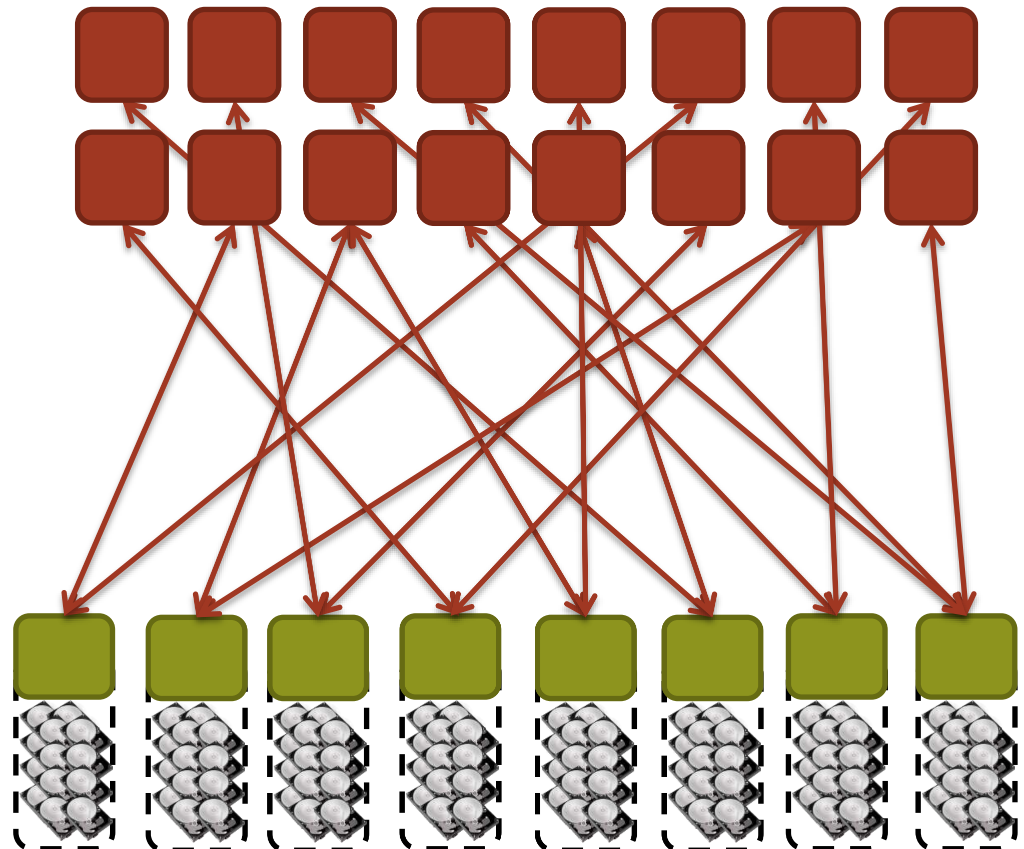
I/O strategies: Spokesperson

- **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**



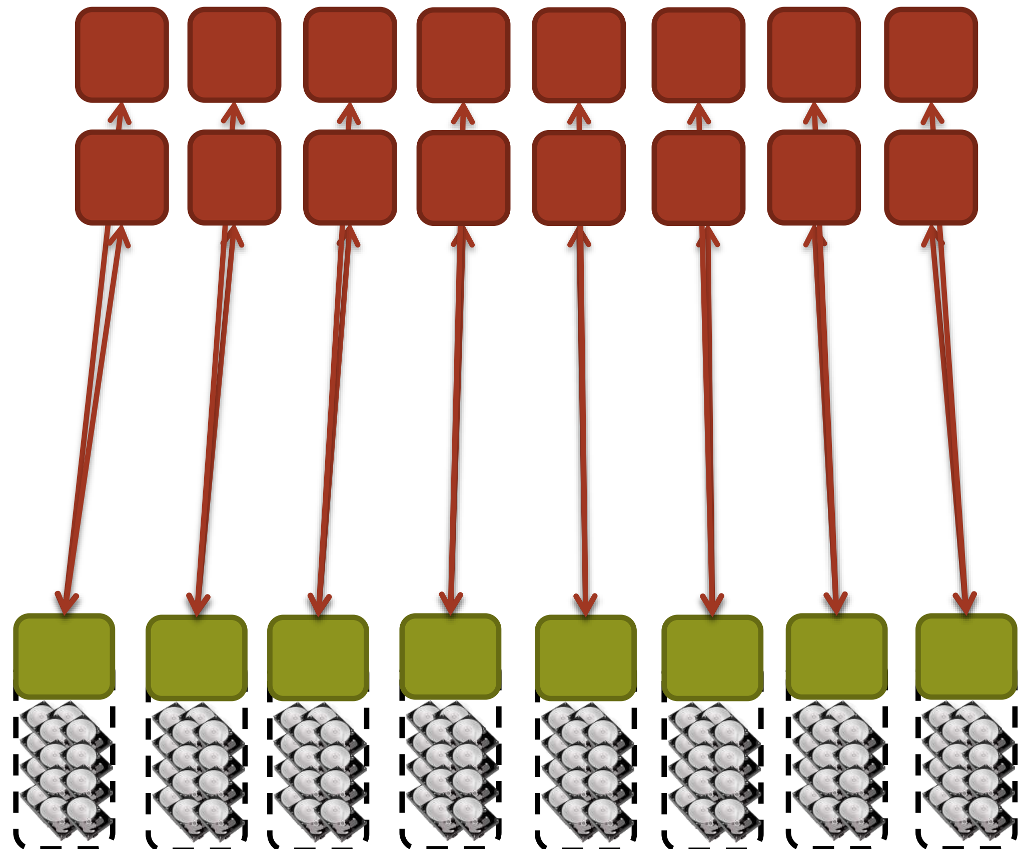
I/O strategies: Multiple Writers – Multiple Files

- **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



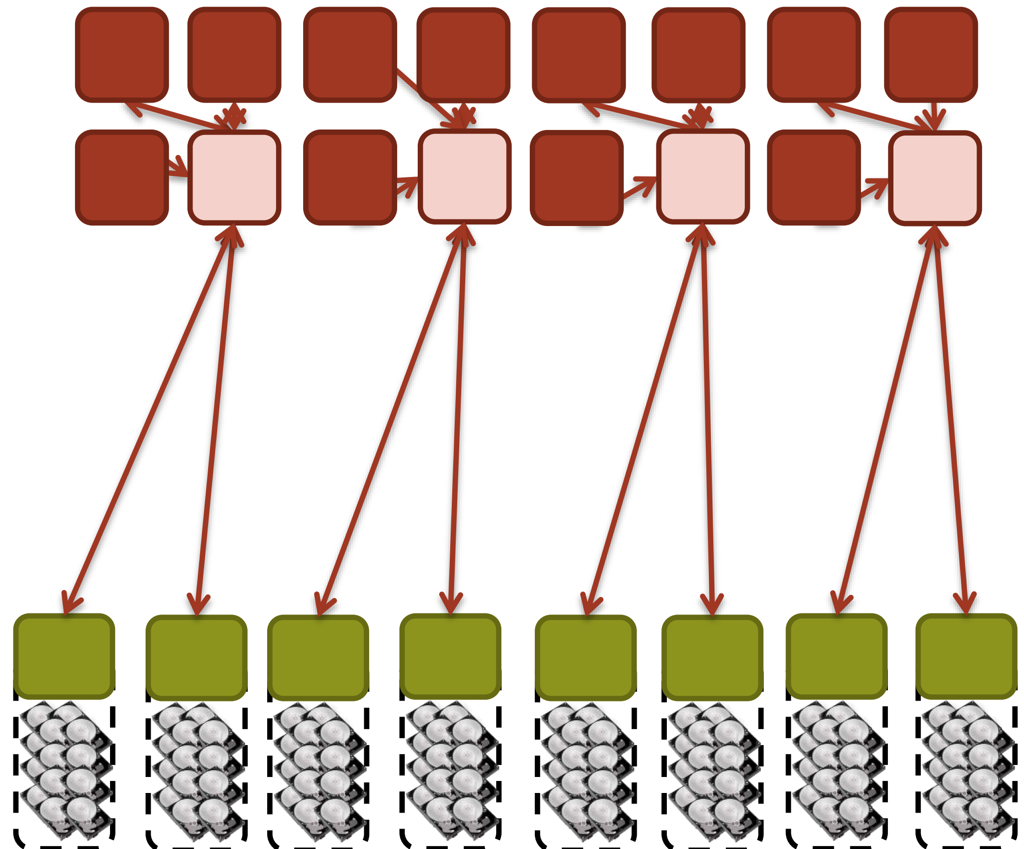
I/O strategies: Multiple Writers – Single File

- **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



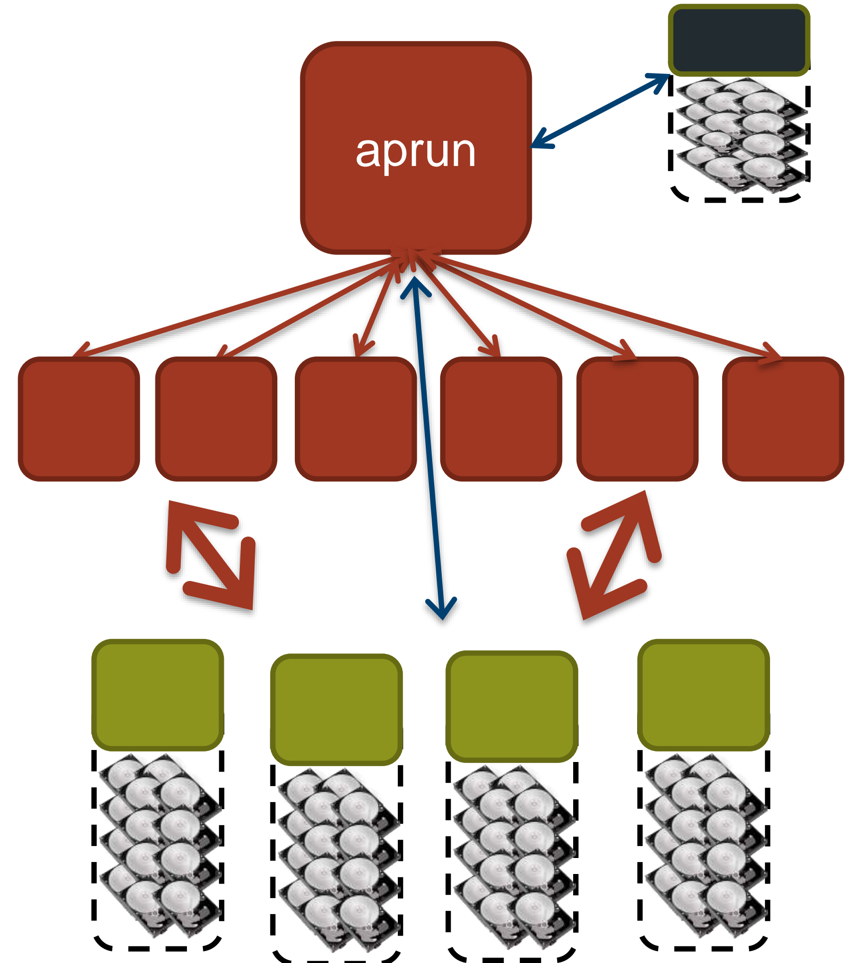
I/O strategies: Collective IO to single or multiple files

- **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.”



Rules for good application I/O performance

CRAY

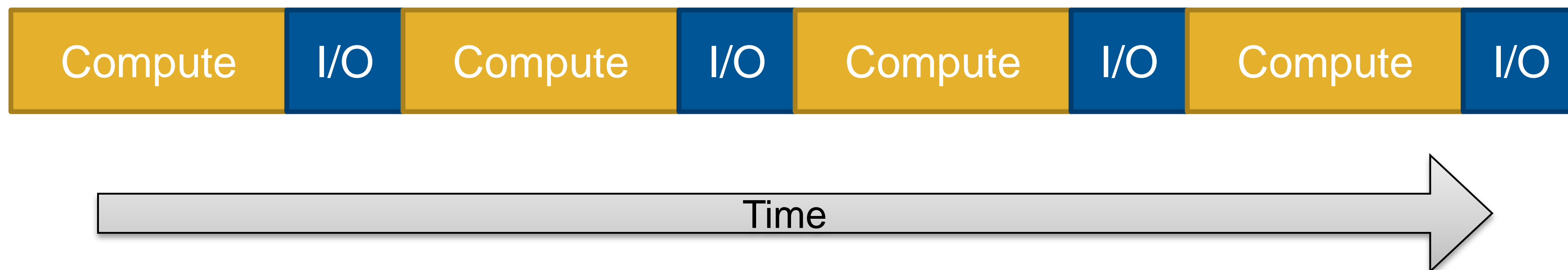
1. Use Parallel I/O
2. Try to hide I/O (asynchronous I/O)
3. Tune filesystem parameters
4. Use I/O buffering for all sequential I/O

I/O performance: to keep in mind

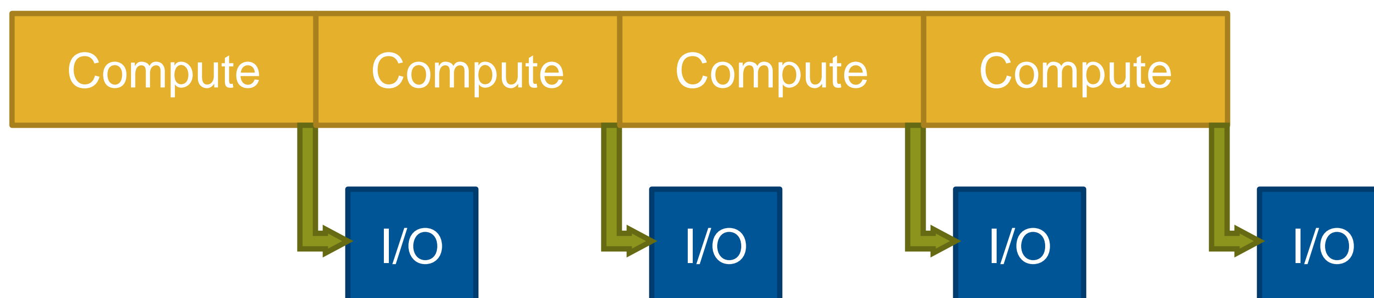
- There is no “One Size Fits All” solution to the I/O problem
- Many I/O patterns work well for some range of parameters
- Bottlenecks in performance can occur in many locations (application and/or filesystem)
- Going to extremes with an I/O pattern will typically lead to problems
- I/O is a shared resource: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

Asynchronous I/O

Standard Sequential I/O



Asynchronous I/O



Asynchronous I/O

CRAY

- **Majority of data is output**
- **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 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

CRAY

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
```

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

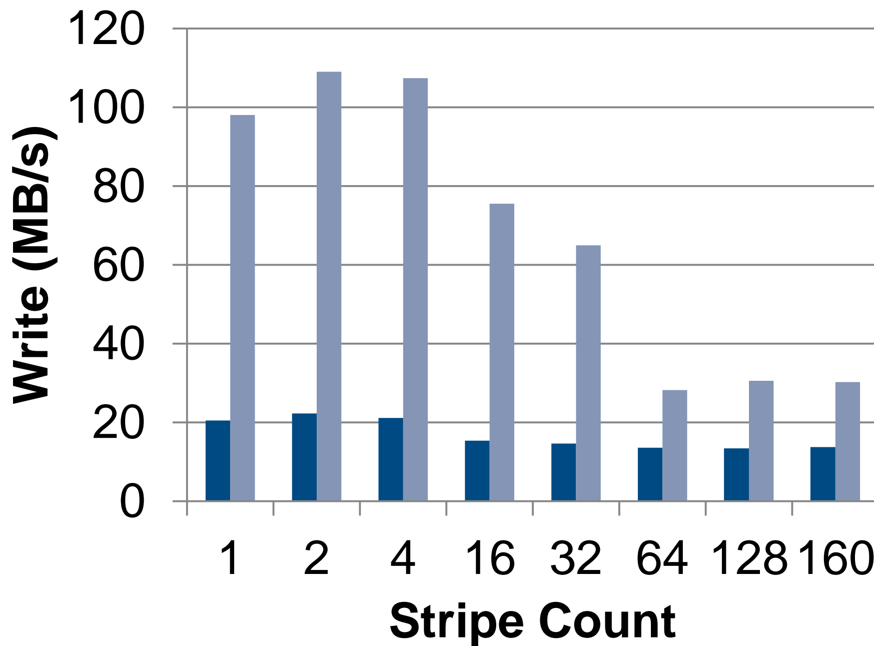
CRAY

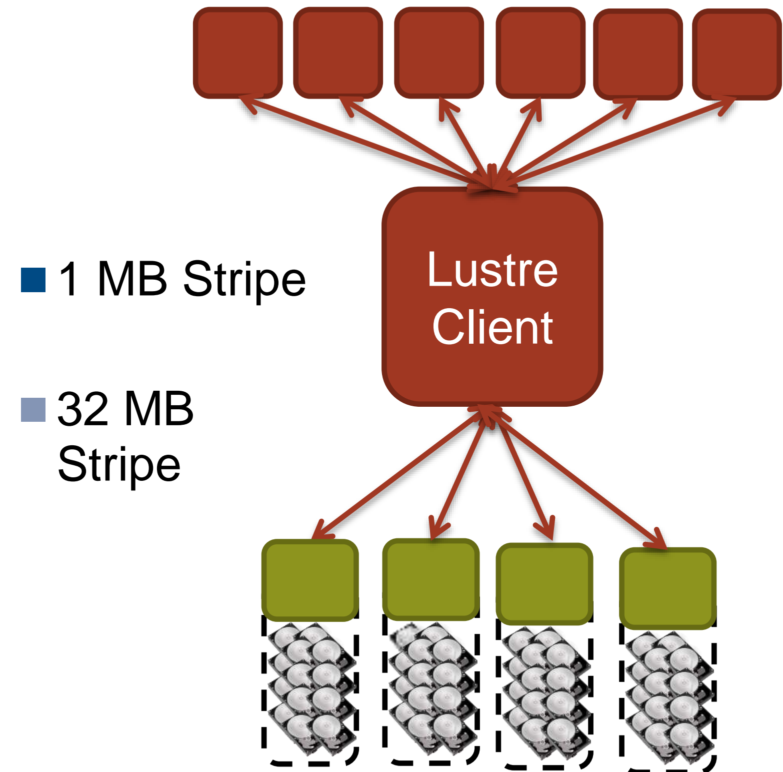
- **Selecting the striping values will have a large impact on the I/O performance of your application**
- **Rule of thumb:**
  1. # files > # OSTs => Set stripe_count=1
     You will reduce the lustre contention and OST file locking this way and gain performance
  2. #files==1 => Set stripe_count=#OSTs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

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

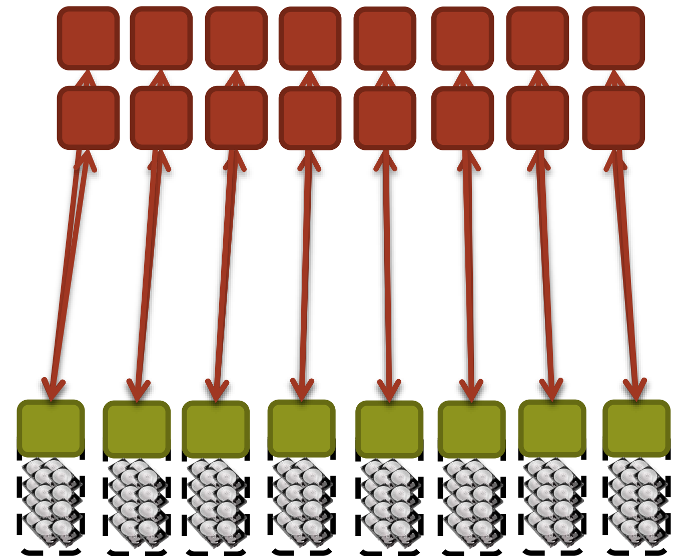




Case Study 2: Parallel I/O into a single file

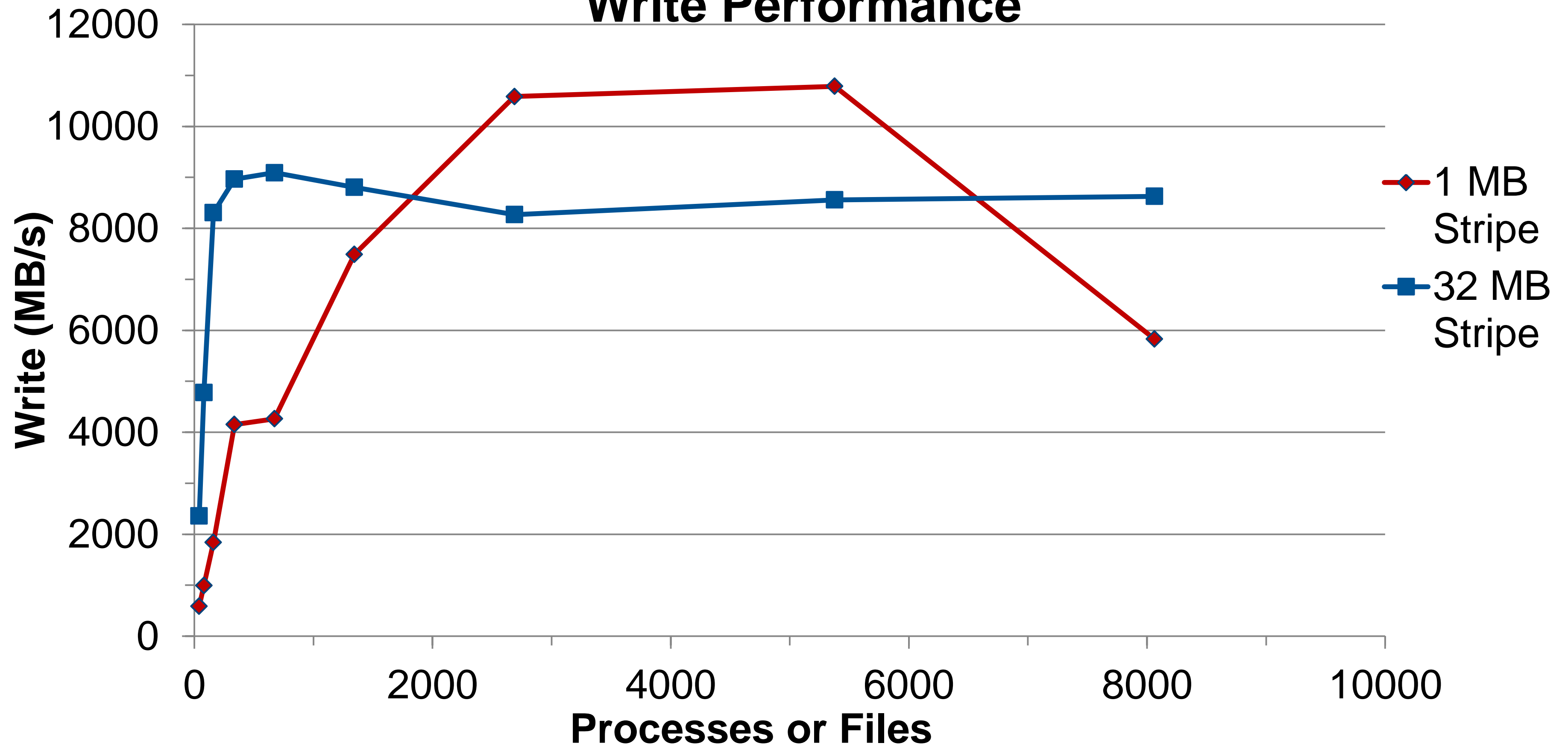
CRAY

- **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.



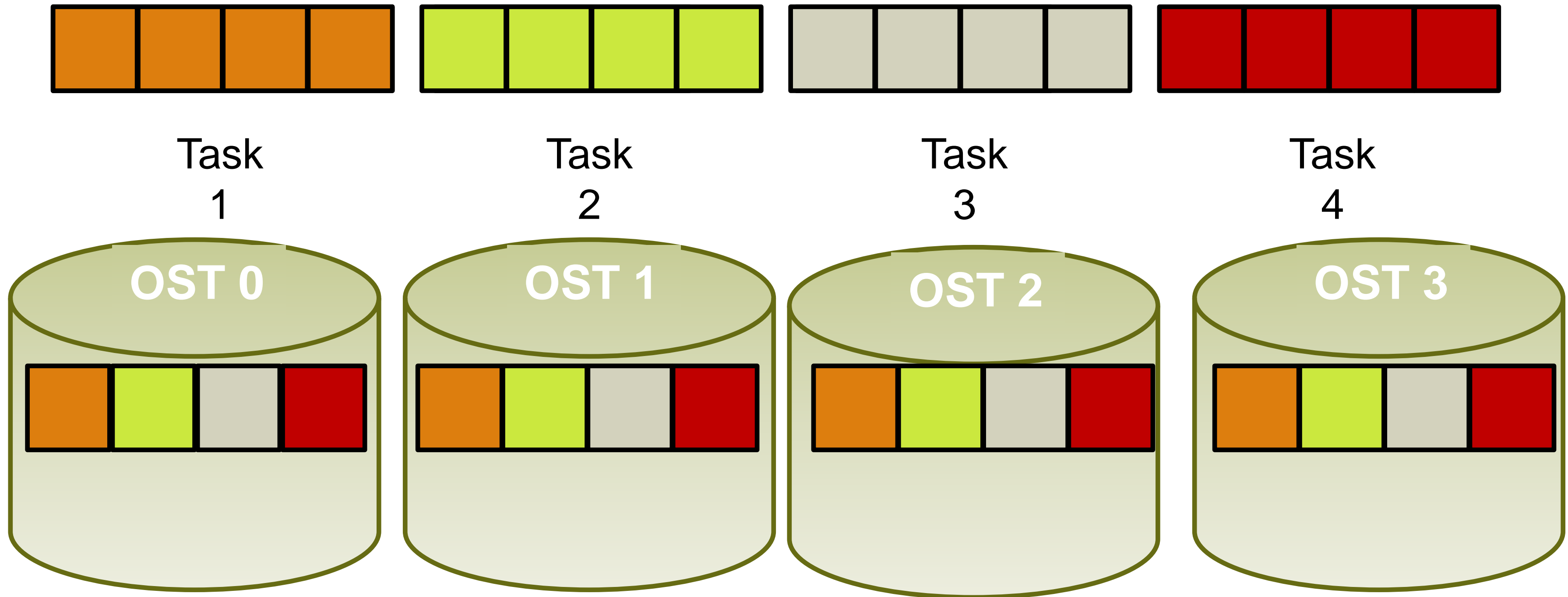
Case Study 3: Single File Per Process

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



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

CRAY

- **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**
  - Take 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
  - 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**
  - Divide the I/O workload equally amongst all aggregators without regard to physical I/O boundaries or Lustre stripes

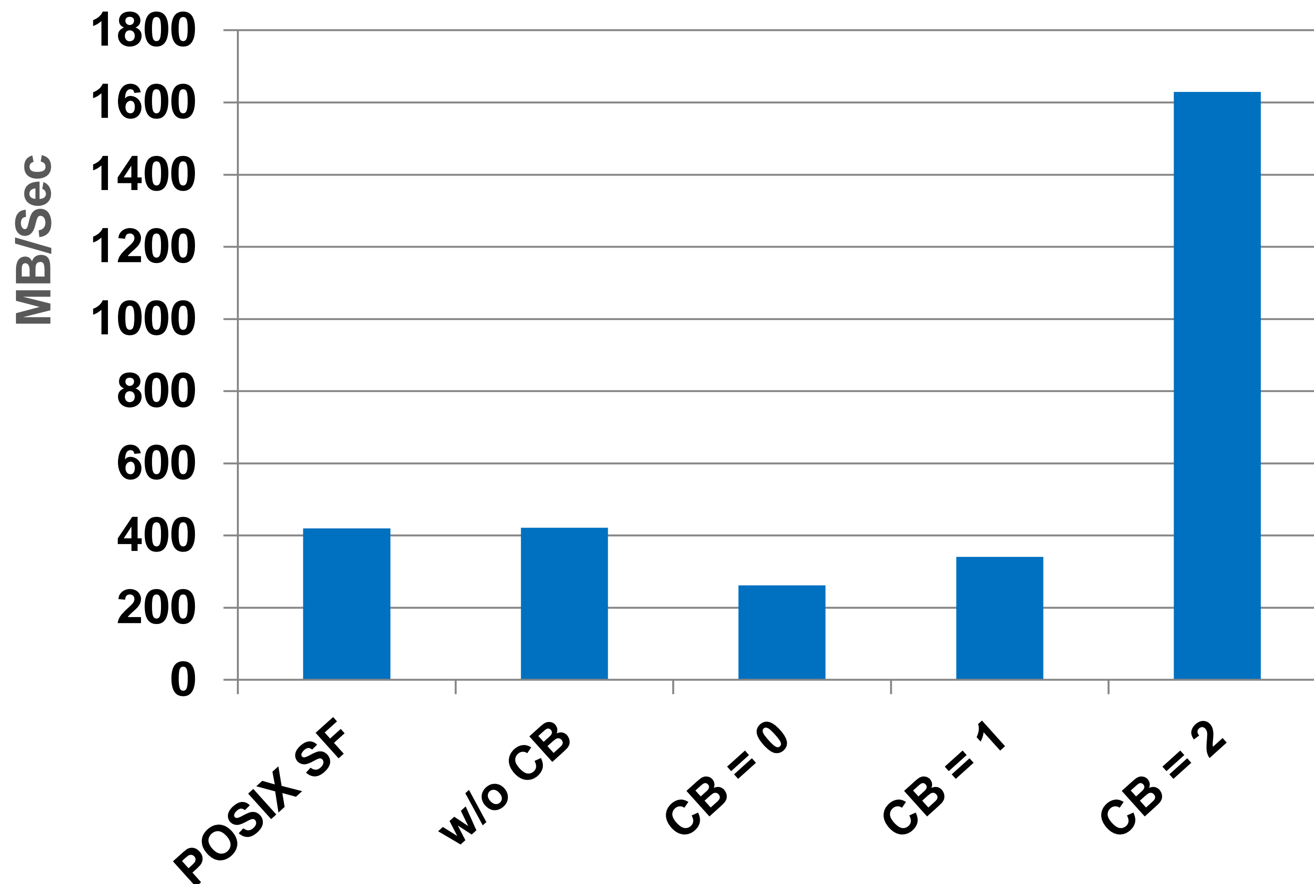
MPI I/O hints (part 1)

- **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

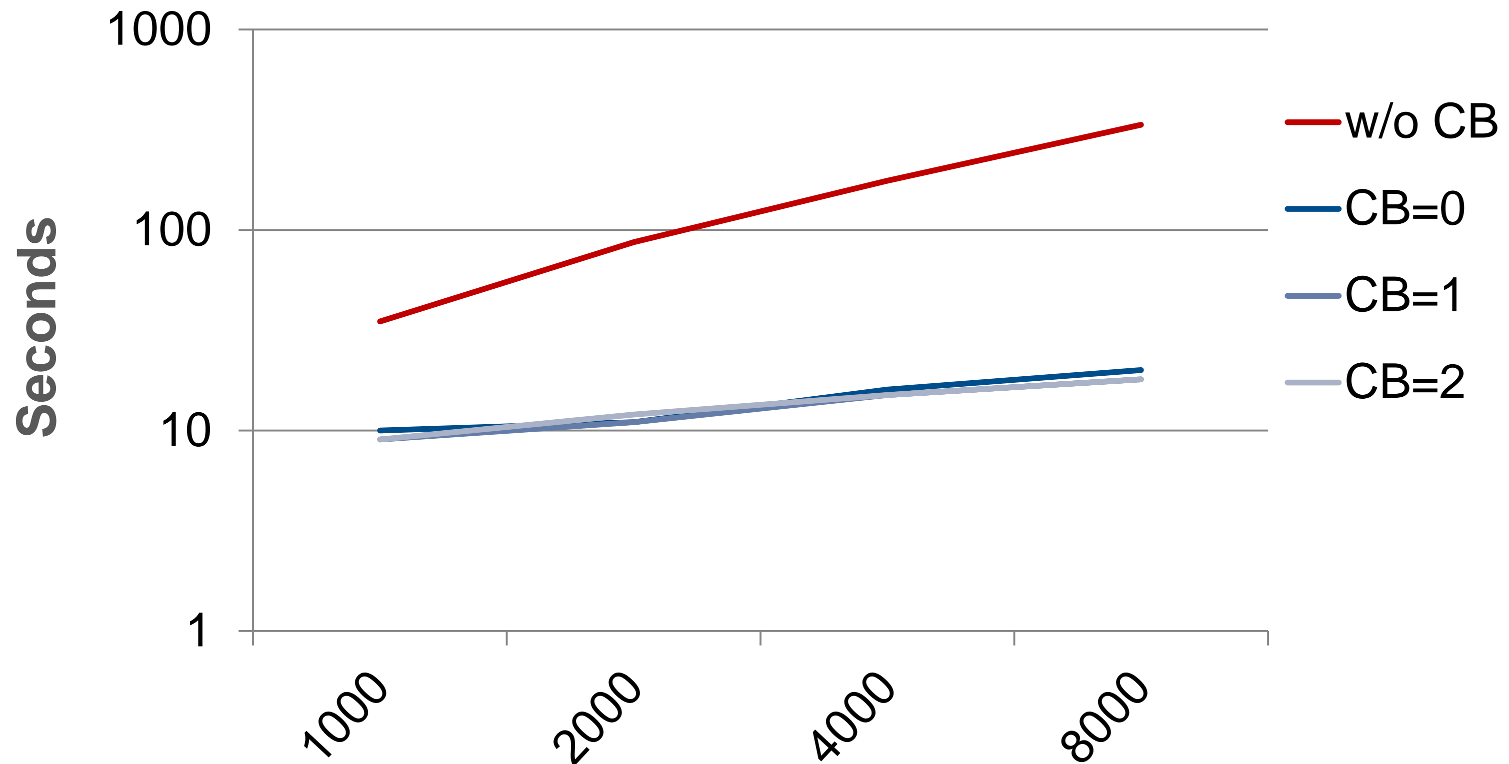
IOR benchmark 1,000,000 bytes

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

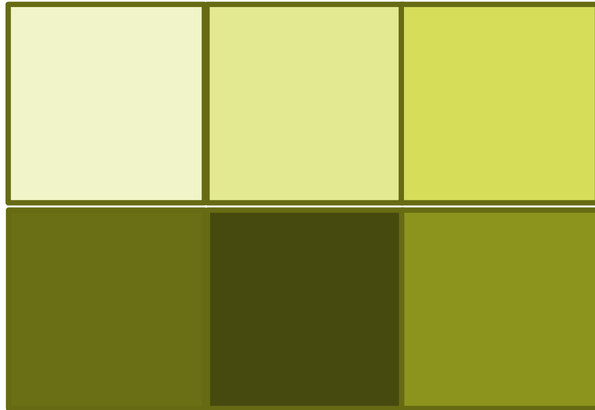


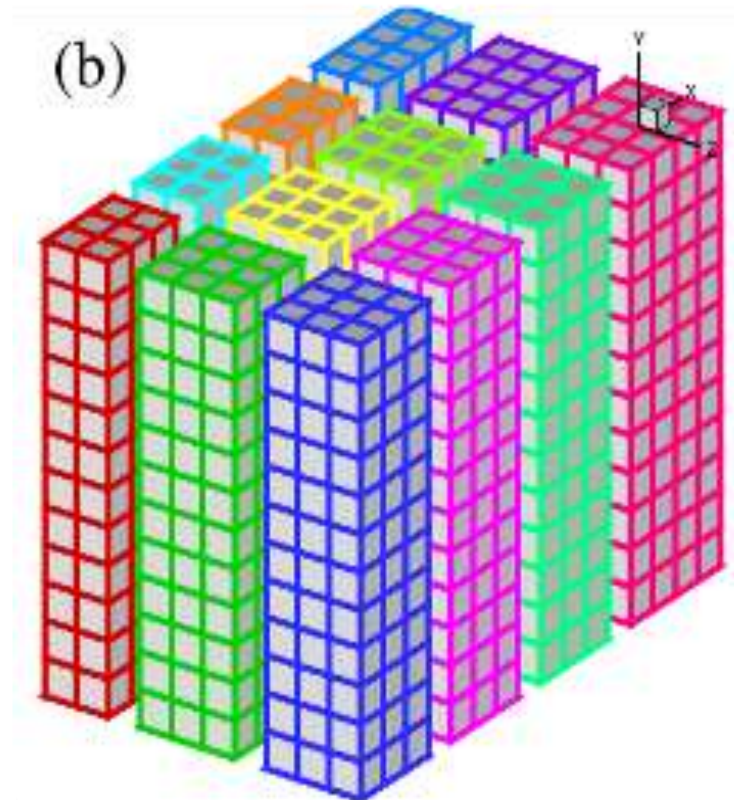
IOBUF

CRAY

- **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/**