A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

Introduction to Parallel I/O

Bilel Hadri
bhadri@utk.edu
NICS Scientific Computing Group

OLCF/NICS Fall Training

October 19th, 2011

Outline

- Introduction to I/O
- Path from Application to File System
- Common I/O Considerations
- I/O Best Practices

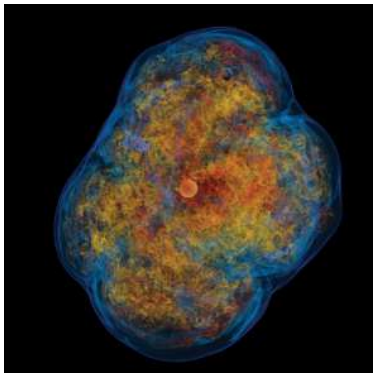
Outline

- **Introduction to I/O**
- Path from Application to File System
- Common I/O Considerations
- I/O Best Practices

Scientific I/O data

I/O is commonly used by scientific applications to achieve goals like:

- storing numerical output from simulations for later analysis
- loading initial conditions or datasets for processing
- checkpointing to files that save the state of an application in case of system failure
- Implement 'out-of-core' techniques for algorithms that process more data than can fit in system memory



HPC systems and I/O

- "A supercomputer is a device for converting a CPU-bound problem into an I/O bound problem." [Ken Batcher]

- Machines consist of three main components:
  - Compute nodes
  - High-speed interconnect
  - I/O infrastructure

- Most optimization work on HPC applications is carried out on
  - Single node performance
  - Network performance (communication)
  - I/O only when it becomes a real problem

The I/O Challenge

- Problems are increasingly computationally challenging
  - Large parallel machines needed to perform calculations
  - Critical to leverage parallelism in all phases
- Data access is a huge challenge
  - Using parallelism to obtain performance
  - Finding usable, efficient, portable interfaces
  - Understanding and tuning I/O
- Data stored in a single simulation for some projects:
  - O(100) TB !!

Why do we need parallel I/O?

- Imagine a 24 hour simulation on 16 cores.
  - 1% of run time is serial I/O.
- You get the compute part of your code to scale to 1024 cores.
  - 64x speedup in compute: I/O is 39% of run time (22'16" in computation and 14'24'' in I/O).
- Parallel I/O is needed to
  - Spend more time doing science
  - Not waste resources
  - Prevent affecting other users

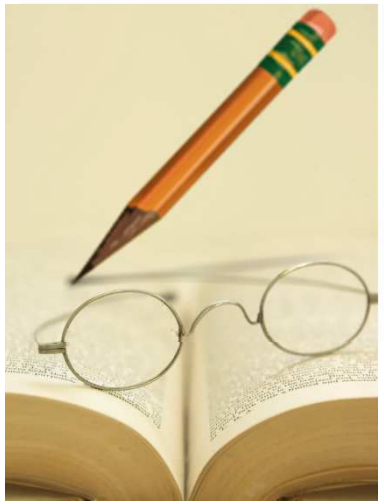
Scalability Limitation of I/O

- I/O subsystems are typically very slow compared to other parts of a supercomputer
  - You can easily saturate the bandwidth

- Once the bandwidth is saturated scaling in I/O stops
  - Adding more compute nodes increases aggregate memory bandwidth and flops/s, but not I/O

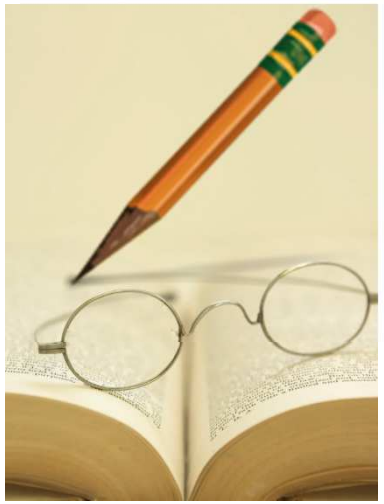
Factors which affect I/O.

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



I/O Performance

- 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.
- Increase performance by decreasing number of I/O operations (latency) and increasing size (bandwidth).



Outline

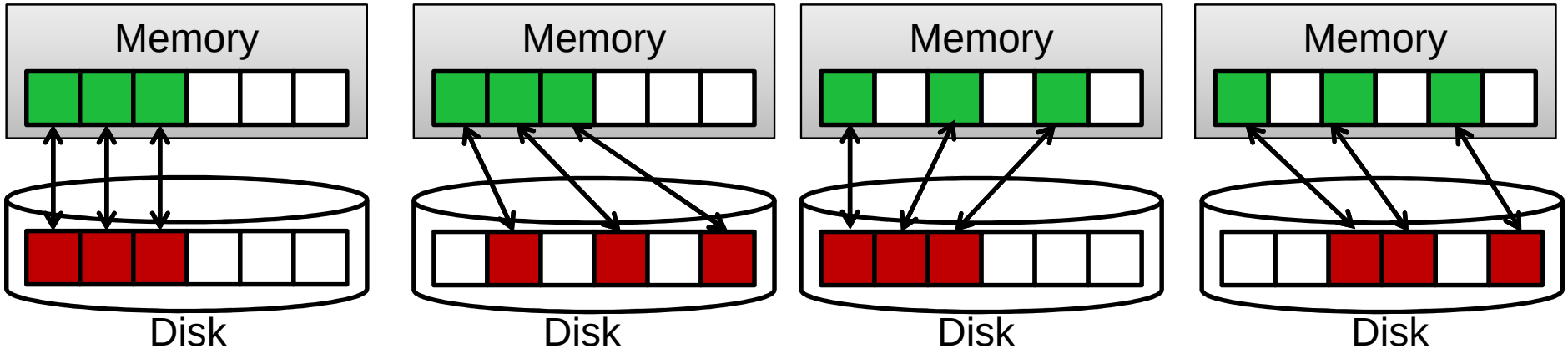
- Introduction to I/O

- **Path from Application to File System**
  - **Data and Performance**
  - **I/O Patterns**
  - **Lustre File System**
  - **I/O Performance Results**

- Common I/O Considerations

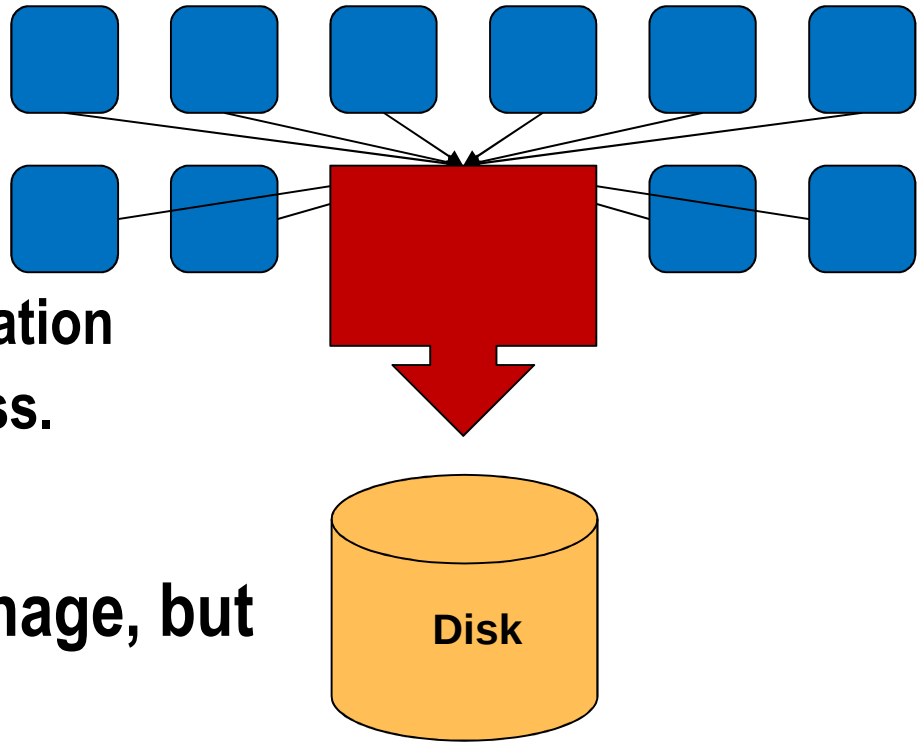
- I/O Best Practices

Data Performance



- Best performance comes from situations when the data is accessed contiguously in memory and on disk.
- Commonly, data access is contiguous in memory but noncontiguous on disk. For example, to reconstruct a global data structure via parallel I/O.
- Sometimes, data access may be contiguous on disk but noncontiguous in memory. For example, writing out the interior of a domain without ghost cells.
- A large impact on I/O performance would be observed if data access was noncontiguous both in memory and on disk.

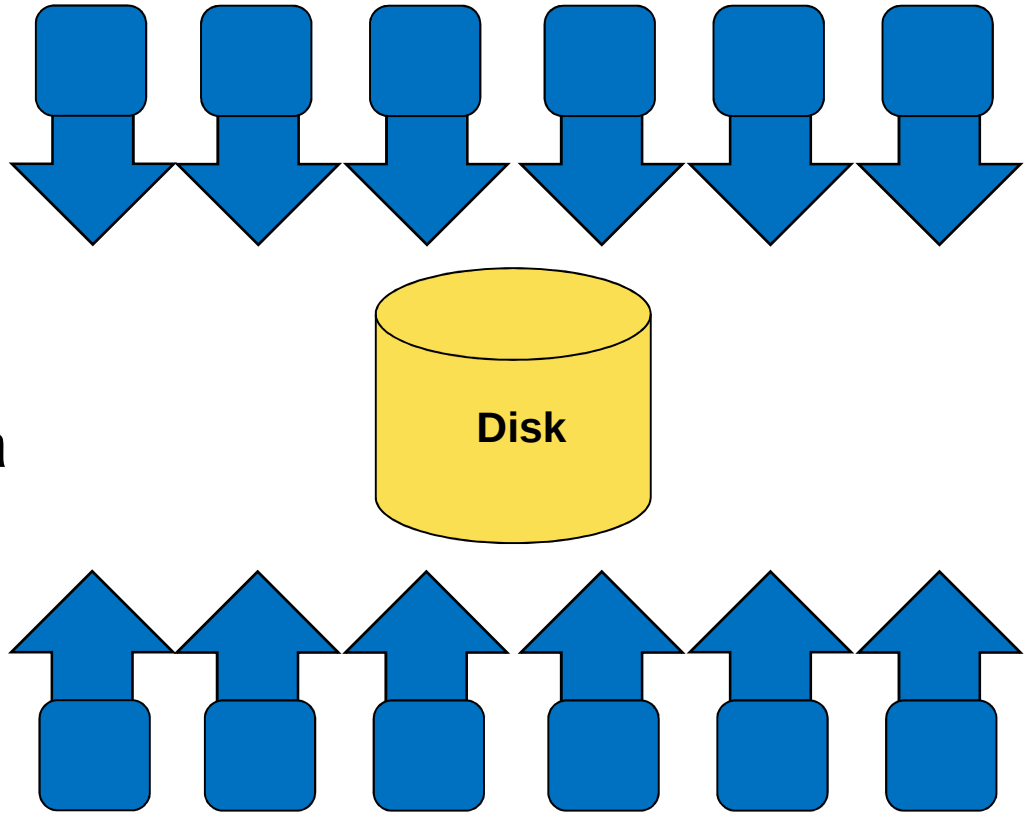
Serial I/O: Spokesperson



- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.
- Simple solution, easy to manage, but
  - Pattern does not scale.
    - Time increases linearly with amount of data.
    - Time increases with number of processes.

Parallel I/O: File-per-Process

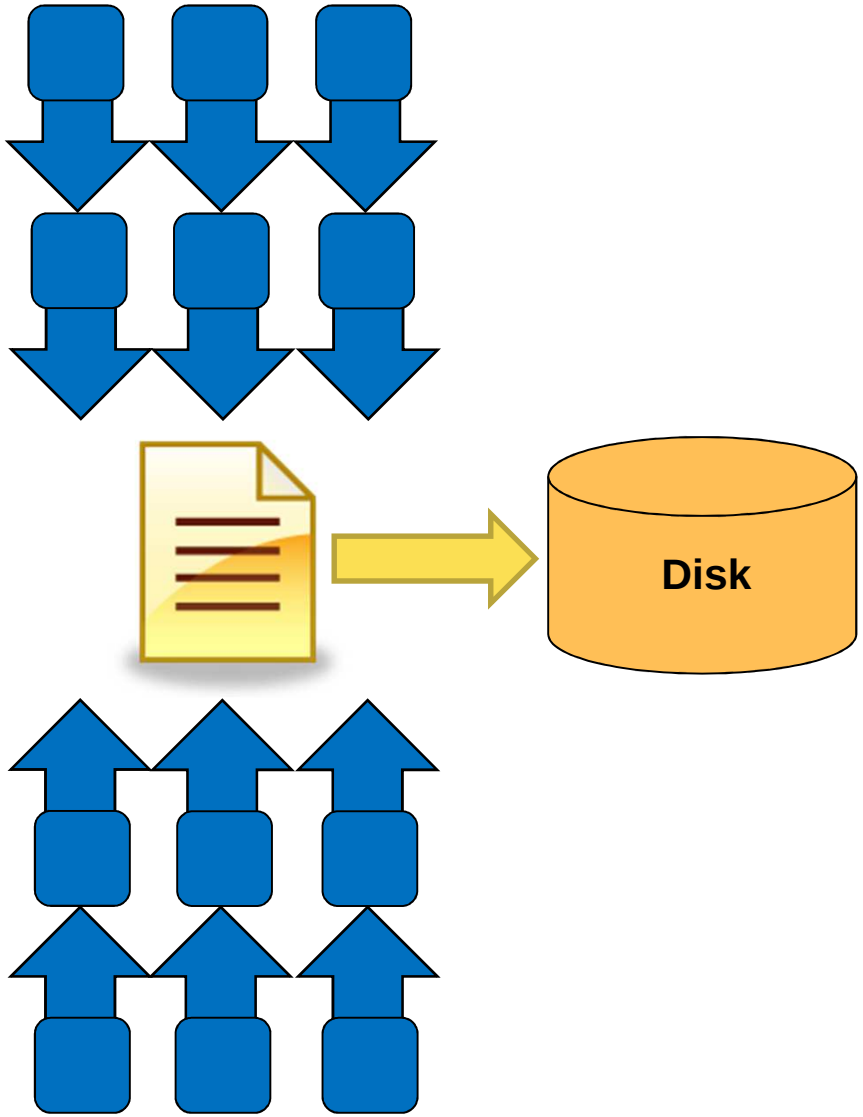
- All processes perform I/O to individual files.
  - Limited by file system.
- Pattern does not scale at large process counts.
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.



NICS

Parallel I/O: Shared File

- 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.



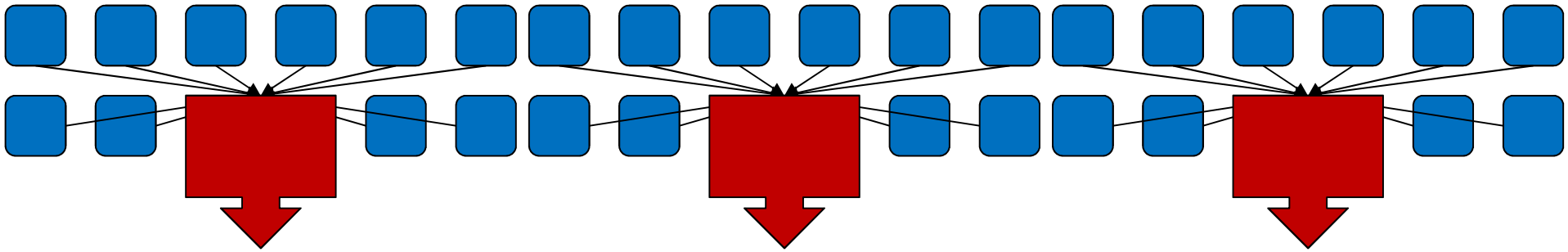
NICS

Pattern Combinations

- Subset of processes which perform I/O.
  - Aggregation of a group of processes 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 shared files

      → increase file system usage.
    - Decreases number of processes which access a shared file

      → decrease file system contention.



Performance Mitigation Strategies

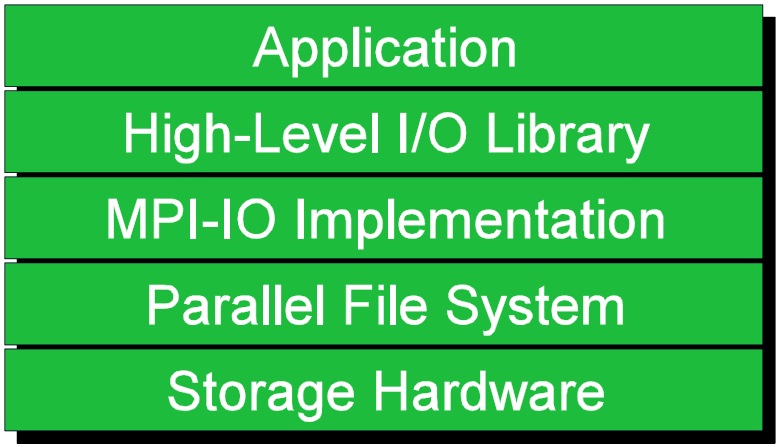
- File-per-process I/O
  - Restrict the number of processes/files written simultaneously. Limits file system limitation.
  - Buffer output to increase the I/O operation size.

- Shared file I/O
  - Restrict the number of processes accessing file simultaneously. Limits file system limitation.
  - Aggregate data to a subset of processes to increase the I/O operation size.
  - Decrease the number of I/O operations by writing/reading strided data.

Parallel I/O Tools

- Collections of system software and libraries have grown up to address I/O issues
  - Parallel file systems
  - MPI-IO
  - High level libraries
- Relationships between these are not always clear.
- Choosing between tools can be difficult.

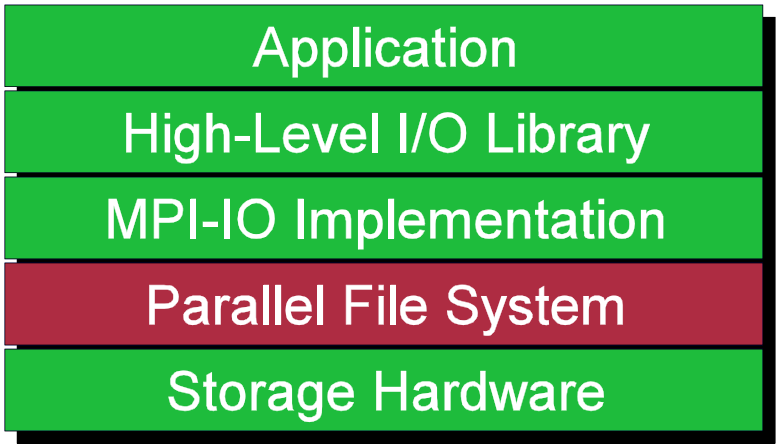
Parallel I/O tools for Computational Science



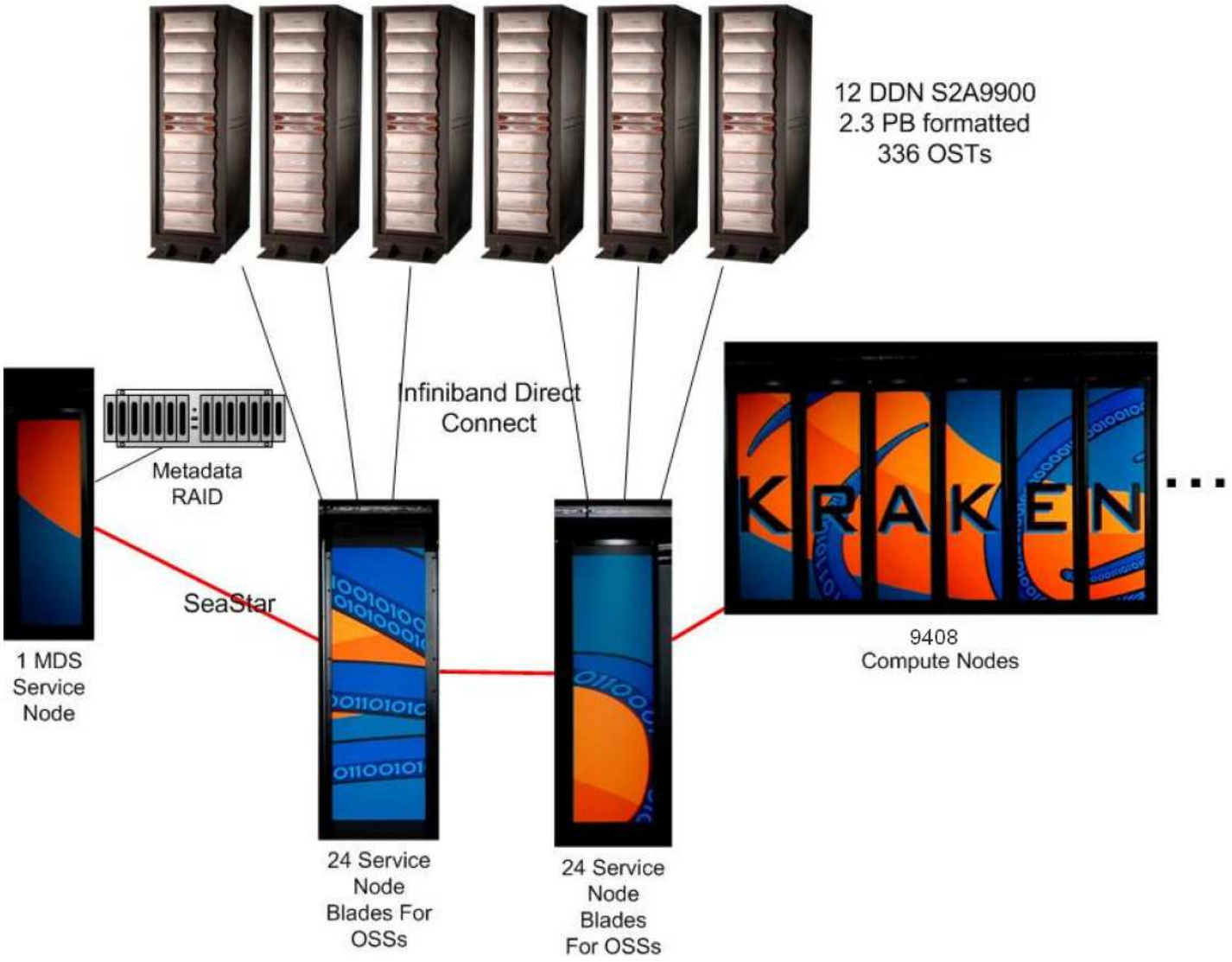
- **Break up support into multiple layers:**
  - **High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF, ADIOS)**
  - **Middleware layer deals with organizing access by many processes (e.g. MPI-IO)**
  - **Parallel file system maintains logical space, provides efficient access to data (e.g. Lustre)**

Parallel File System

- Manage storage hardware
  - Present single view
  - Focus on concurrent, independent access
  - Transparent : files accessed over the network can be treated the same as files on local disk by programs and users
  - Scalable

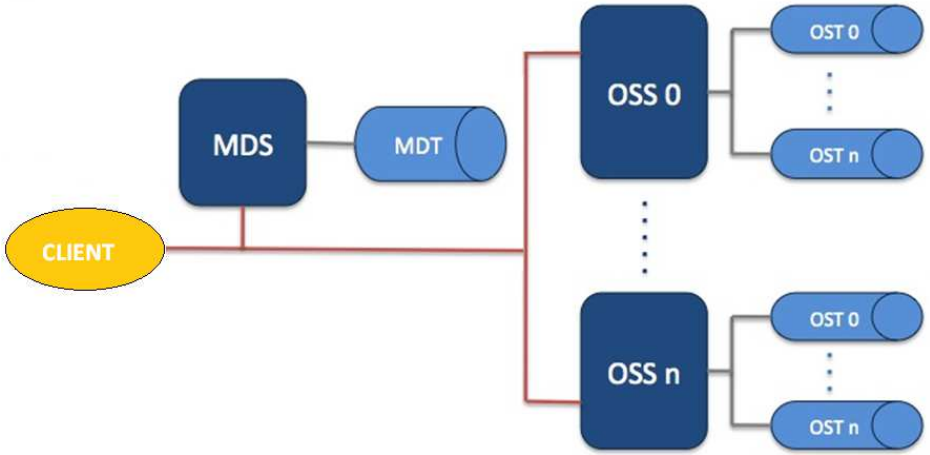


Kraken Lustre Overview



File I/O: Lustre File System

- Metadata Server (MDS) makes metadata stored in the MDT(Metadata Target) available to Lustre clients.
  - The MDS opens and closes files and stores directory and file Metadata such as file ownership, timestamps, and access permissions on the MDT.
  - Each MDS manages the names and directories in the Lustre file system and provides network request handling for the MDT.
- Object Storage Server(OSS) provides file service, and network request handling for one or more local OSTs.
- Object Storage Target (OST) stores file data (chunks of files).

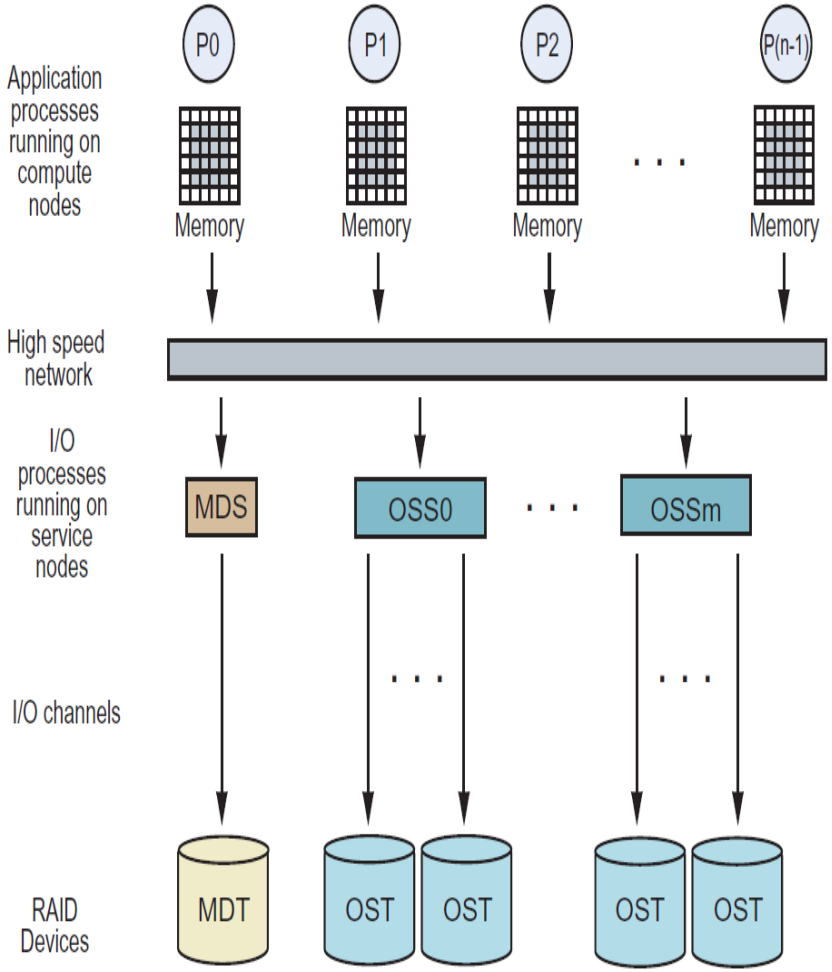


NICS

Lustre

- Once a file is created, write operations take place directly between compute node processes (P0, P1, ...) and Lustre object storage targets (OSTs), going through the OSSs and bypassing the MDS.

- For read operations, file data flows from the OSTs to memory. Each OST and MDT maps to a distinct subset of the RAID devices.



Striping: Storing a single file across multiple OSTs

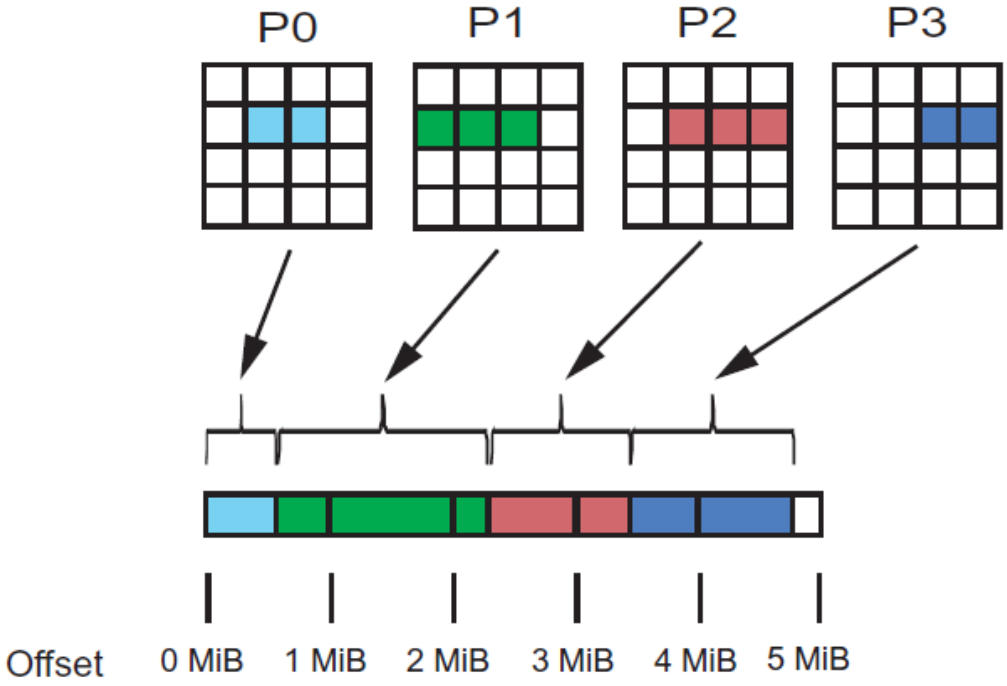
- A single file may be stripped across one or more OSTs (chunks of the file will exist on more than one OST).

- Advantages :
  - an increase in the bandwidth available when accessing the file
  - an increase in the available disk space for storing the file.

- Disadvantage:
  - increased overhead due to network operations and server contention

→ Lustre file system allows users to specify the striping policy for each file or directory of files using the lfs utility

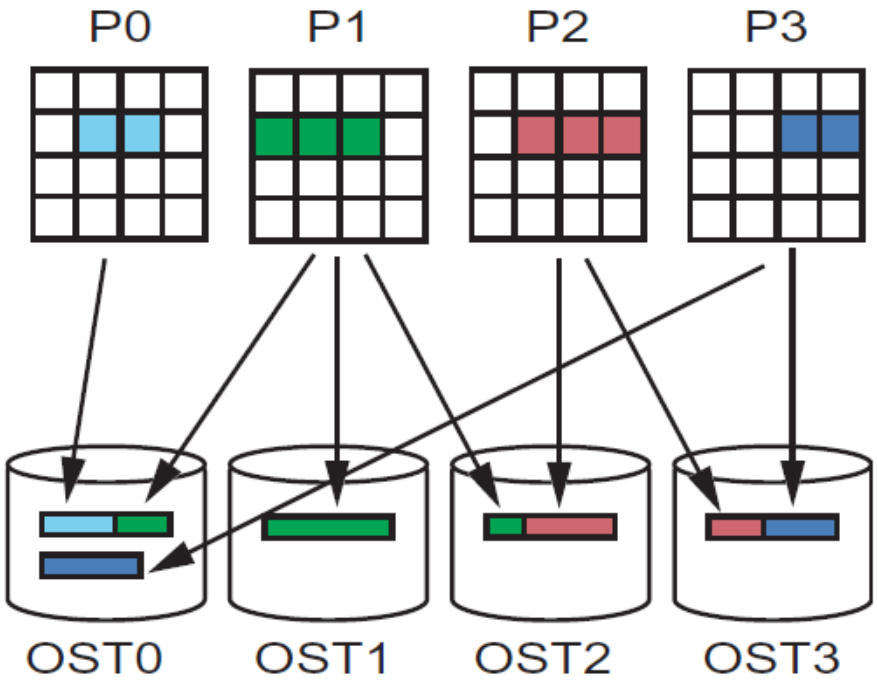
File Striping: Physical and Logical Views



Four application processes write a variable amount of data sequentially within a shared file. This shared file is striped over 4 OSTs with 1 MB stripe sizes.

This write operation is not stripe aligned therefore some processes write their data to stripes used by other processes. Some stripes are accessed by more than one process

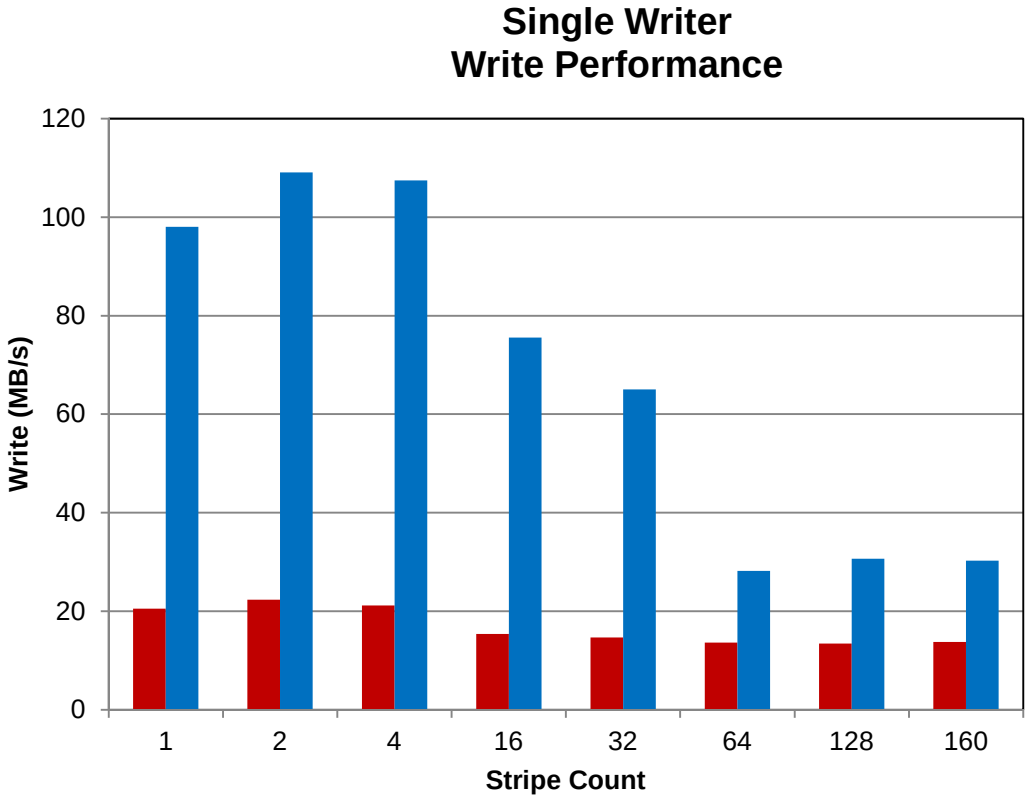
→ May cause contention !

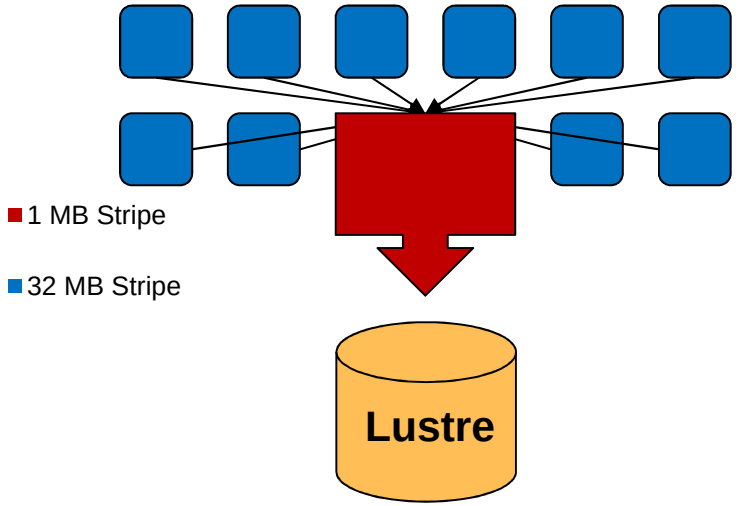


OSTs are accessed by variable numbers of processes (3 OST0, 1 OST1, 2 OST2 and 2 OST3).

Single writer performance and Lustre

- 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

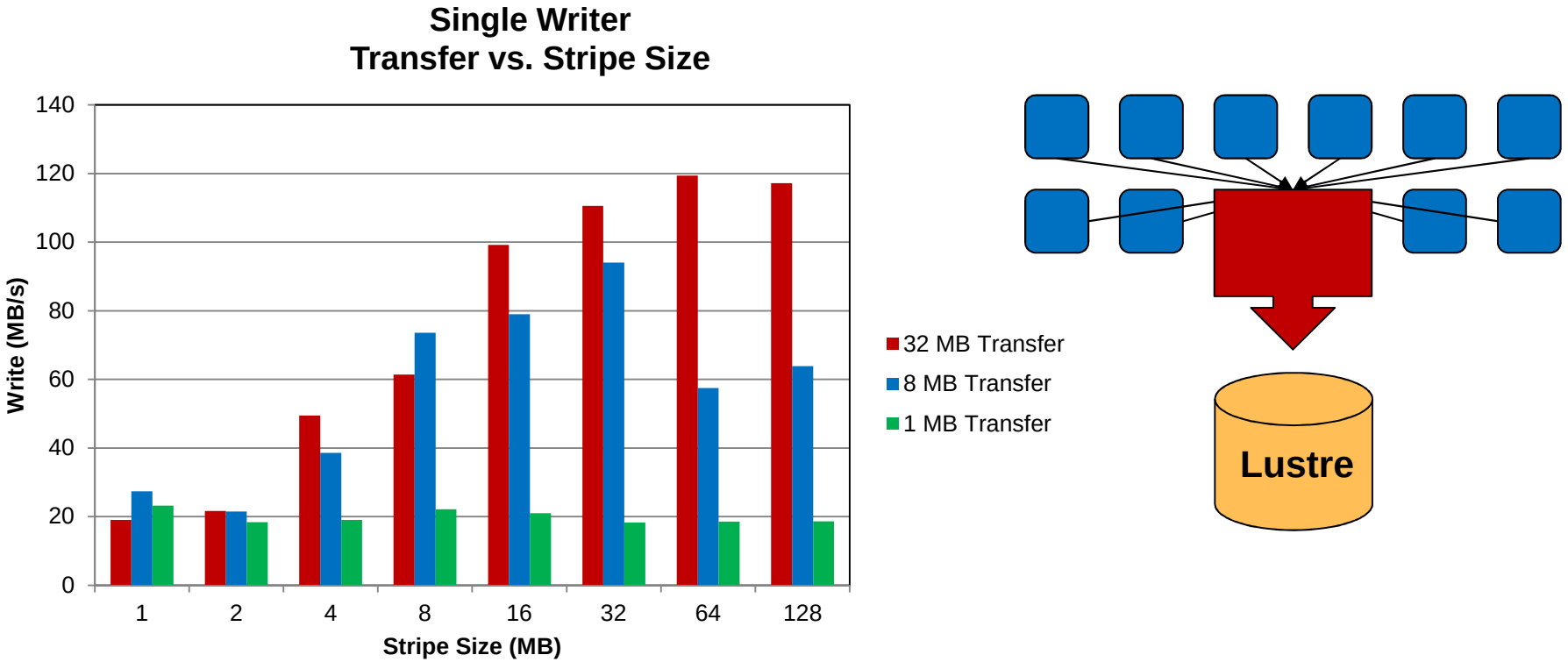




→ Using more OSTs does not increase write performance. (Parallelism in Lustre cannot be exploit)

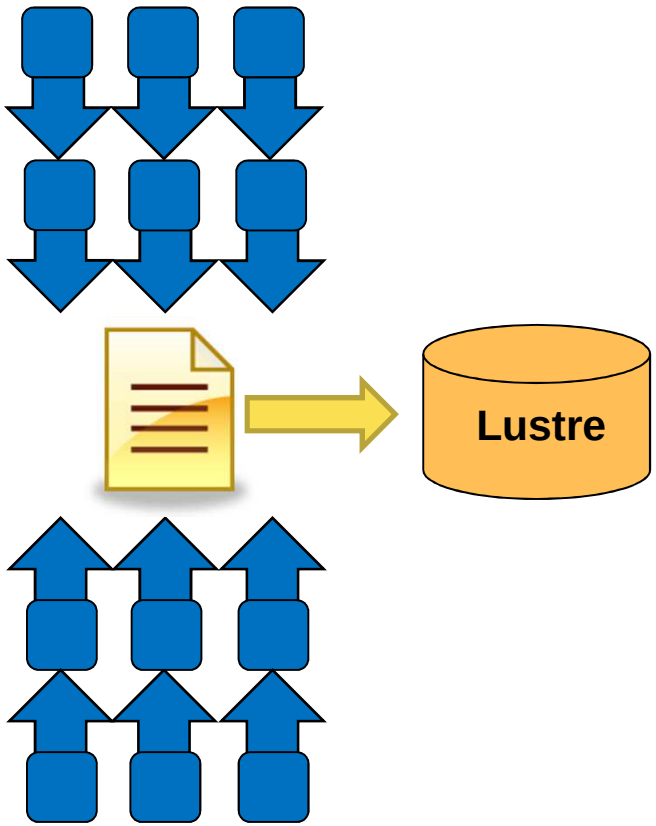
Stripe size and I/O Operation size

- **Single OST, 256 MB File Size**
  - **Performance can be limited by the process (transfer size) or file system (stripe size). Either can become a limiting factor in write performance.**



→ The best performance is obtained in each case when the I/O operation and stripe sizes are similar.
→ Larger I/O operations and matching Lustre stripe setting may improve performance (reduces the latency of I/O op.)

Single Shared Files and Lustre Stripes

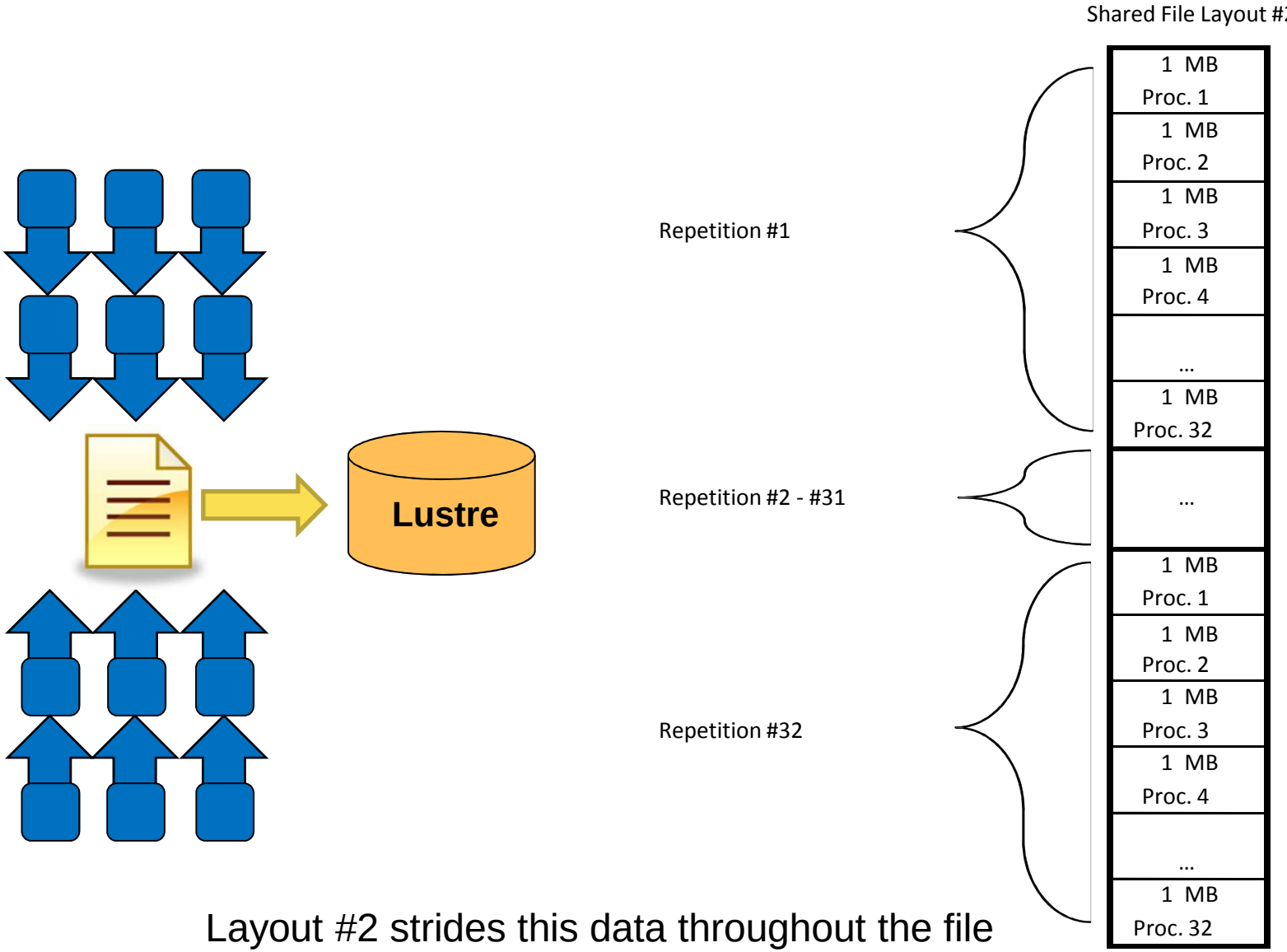


Shared File Layout #1

32 MB Proc. 1
32 MB Proc. 2
32 MB Proc. 3
32 MB Proc. 4
…
32 MB Proc. 32

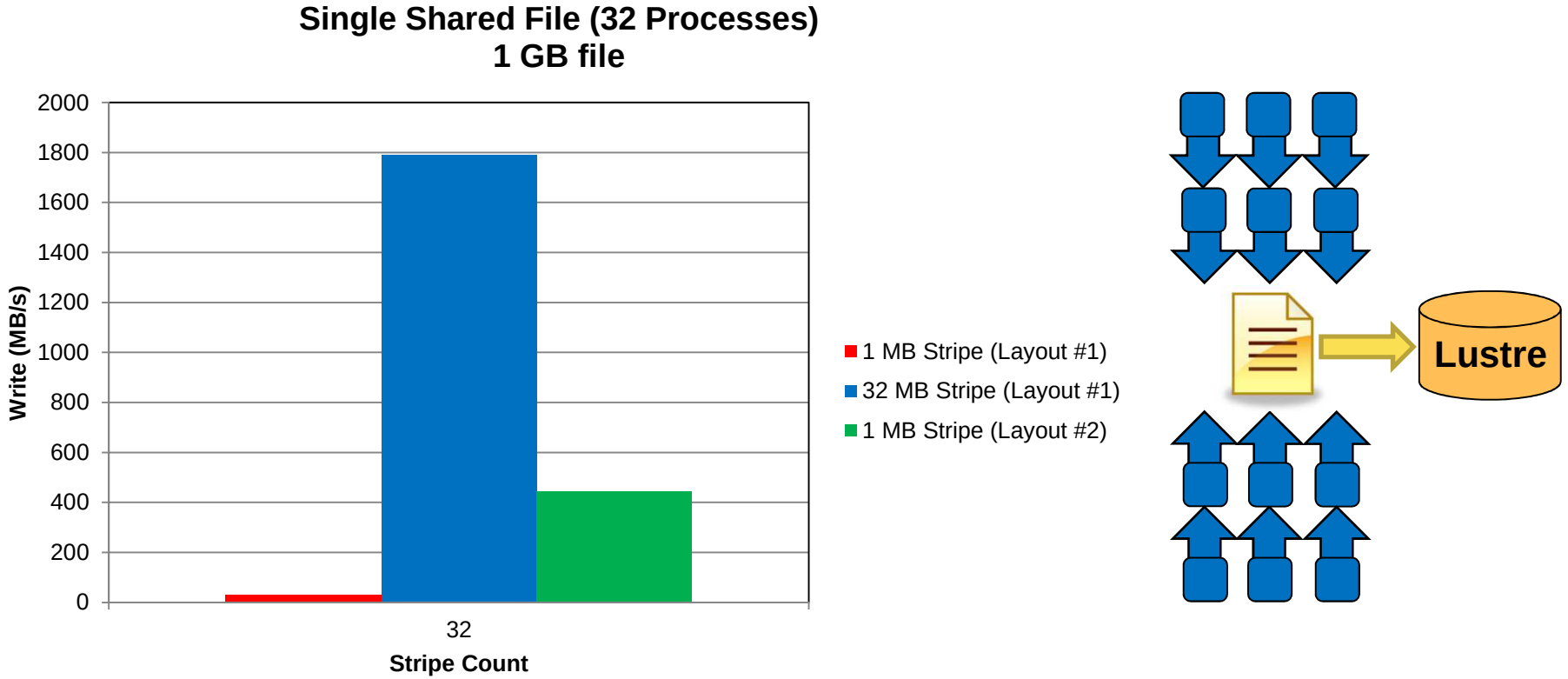
Layout #1 keeps data from a process in a contiguous block

Single Shared Files and Lustre Stripes



Layout #2 strides this data throughout the file

File Layout and Lustre Stripe Pattern



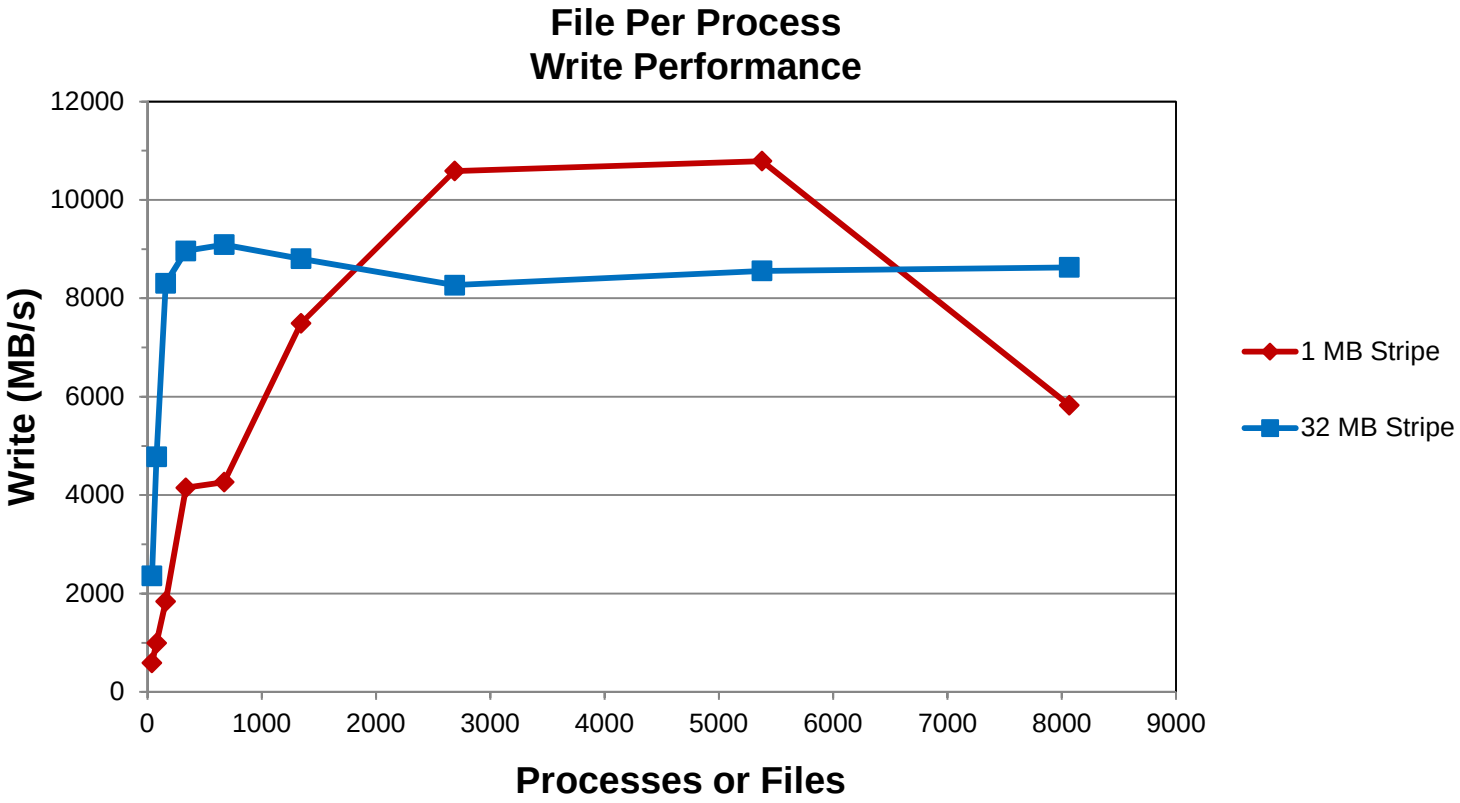
→ A 1 MB stripe size on Layout #1 results in the lowest performance due to OST contention. Each OST is accessed by every process. (31.18 MB/s)

→ The highest performance is seen from a 32 MB stripe size on Layout #1. Each OST is accessed by only one process. (1788,98 MB/s)

→ A 1 MB stripe size gives better performance with Layout #2. Each OST is accessed by only one process. However, the overall performance is lower due to the increased latency in the write (smaller I/O operations). (442.63MB/s)

Scalability: File Per Process

- 128 MB per file and a 32 MB Transfer size

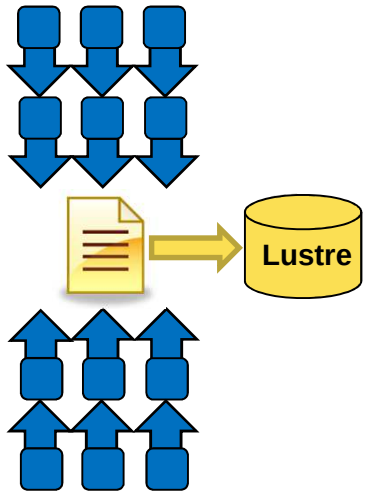


→ Performance increases as the number of processes/files increases until OST and metadata contention hinder performance improvements.

→ At large process counts (large number of files) metadata operations may hinder overall performance due to OSS and OST contention.

Case Study: Parallel I/O

- 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.



I/O Scalabity

- Lustre
  - Minimize contention for file system resources.
  - A process should not access more than one or two OSTs.
  - Decrease the number of I/O operations (latency).
  - Increase the size of I/O operations (bandwidth).

Scalability

- Serial I/O:
  - Is not scalable. Limited by single process which performs I/O.

- File per Process
  - Limited at large process/file counts by:
    - Metadata Operations
    - File System Contention

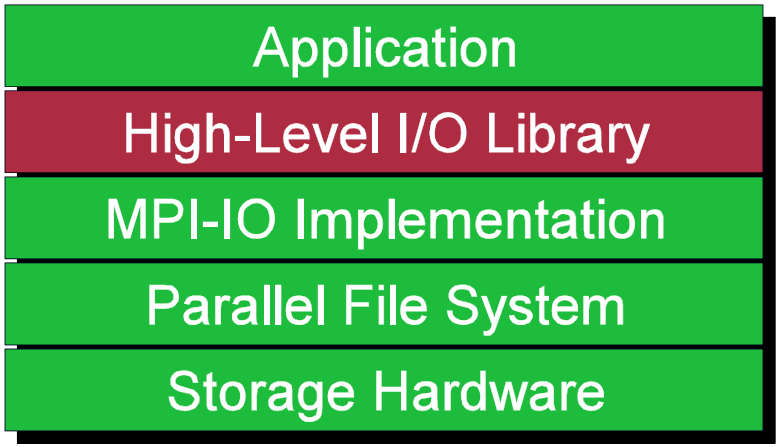
- Single Shared File
  - Limited at large process counts by file system contention.

Outline

- Introduction to I/O
- Path from Application to File System
- **Common I/O Considerations**
  - **I/O libraries**
  - **MPI I/O usage**
  - **Buffered I/O**
- I/O Best Practices

High Level Libraries

- Provide an appropriate abstraction for domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Self-describing, structured file format
- Map to middleware interface
  - Encourage collective I/O
- Provide optimizations that middleware cannot



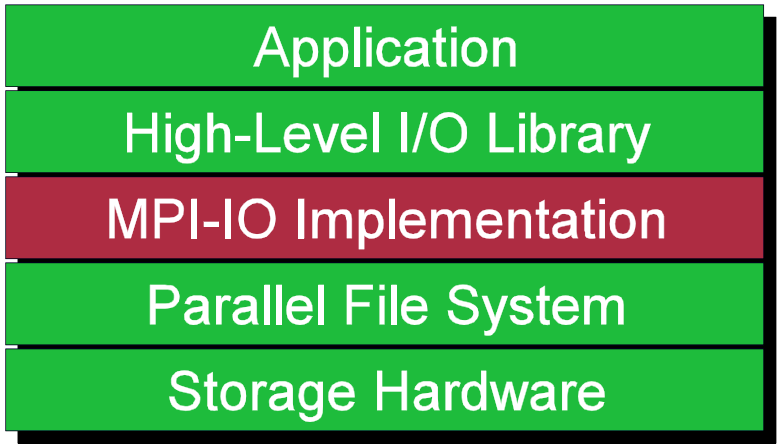
POSIX

- POSIX interface is a useful, ubiquitous interface for building basic I/O tools.
  - Standard I/O interface across many platforms.
    - open, read/write, close functions in C/C++/Fortran
  - Mechanism almost all serial applications use to perform I/O
  - No way of describing collective access
- No constructs useful for parallel I/O.
- Should not be used in parallel applications if performance is desired !

I/O Libraries

- One of the most used libraries on Jaguar and Kraken.
- Many I/O libraries such as HDF5 , Parallel NetCDF and ADIOS are built atop MPI-IO.
- Such libraries are abstractions from MPI-IO.
- Such implementations allow for higher information propagation to MPI-IO (without user intervention).

MPI-I/O: the Basics



- MPI-IO provides a low-level interface to carrying out parallel I/O
- The MPI-IO API has a large number of routines.
- As MPI-IO is part of MPI, you simply compile and link as you would any normal MPI program.
- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules

I/O 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().
  - Similar to POSIX I/O, but supports derived datatypes and thus noncontiguous data and nonuniform strides and can take advantages 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

MPI I/O: Simple C example- (using individual pointers)

```
/* Open the file */
MPI_File_open(MPI_COMM_WORLD,'file', MPI_MODE_RDONLY, MPI_INFO_NULL,&fh);

/* Get the size of file */
MPI_File_get_size(fh, &filesize);
bufsize = filesize/nprocs;
nints = bufsize/sizeof(int);

/* points to the position in the file where each process will start reading data */
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);

/* Each process read in data from the file */
MPI_File_read(fh, buf, nints, MPI_INT, &status);

/* Close the file */
MPI_File_close(&fh);
```

MPI I/O: Fortran example- (using explicit offsets)

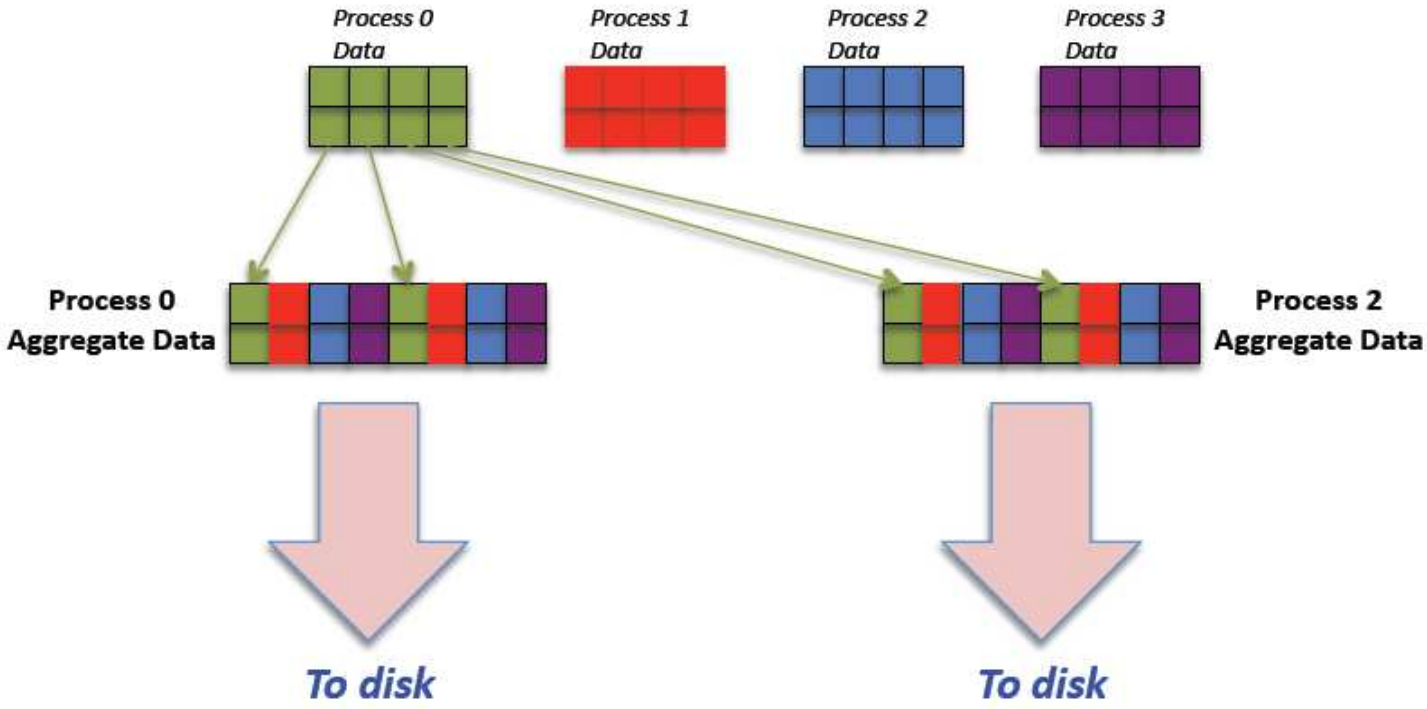
```
! Open the file
call MPI_FILE_OPEN(MPI_COMM_WORLD, ‘file', MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)

! Get the size of file
call MPI_File_get_size(fh, filesize,ierr);
nints = filesize/ (nprocs*INTSIZE)
offset = rank * nints * INTSIZE

! Each process read in data from the file
call MPI_FILE_READ_AT(fh, offset, buf, nints, MPI_INTEGER, status, ierr)

! Close the file
call MPI_File_close(fh,ierr);
```

Collective I/O with MPI-IO

- MPI_File_read[write]_all, MPI_File_read[write]_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 Collective Writes and Optimizations

- When writing in collective mode, the MPI library carries out a number of optimizations
  - It uses fewer processes to actually do the writing
    - Typically one per node
  - It aggregates data in appropriate chunks before writing



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
  - 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.
  - man mpi for more information

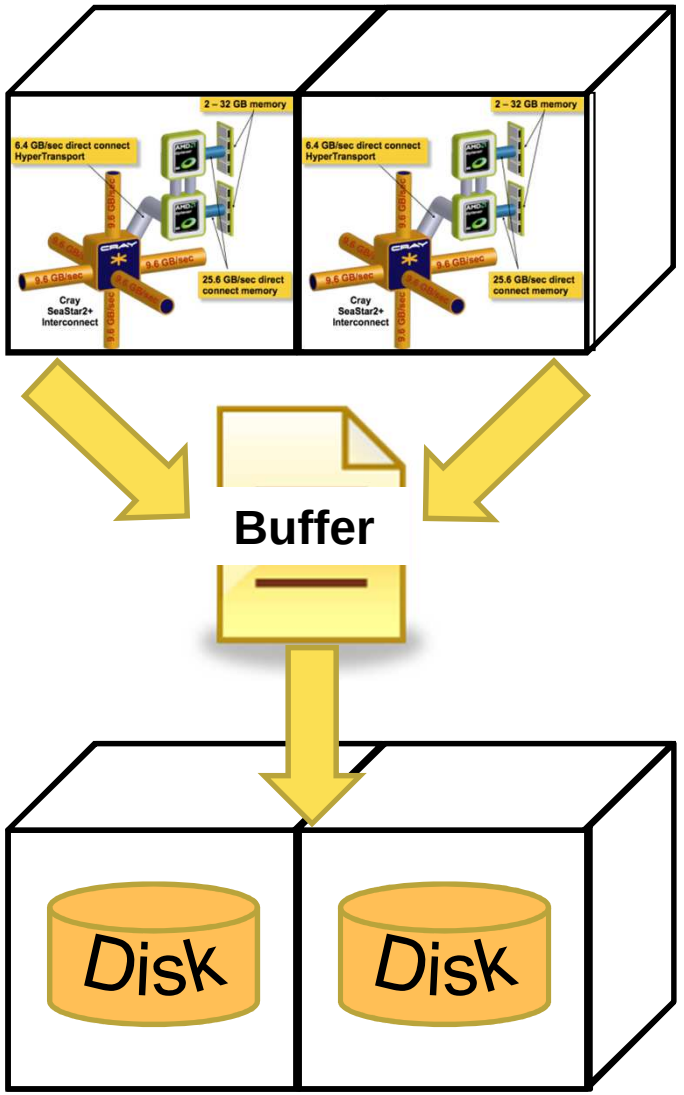
MPI-IO_HINTS

- MPI-IO are generally implementation specific. Below are options from the Cray XT5. (partial)
  - striping_factor (Lustre stripe count)
  - striping_unit (Lustre stripe size)
  - cb_buffer_size (Size of Collective buffering buffer)
  - cb_nodes (Number of aggregators for Collective buffering)
  - ind_rd_buffer_size (Size of Read buffer for Data sieving)
  - ind_wr_buffer_size (Size of Write buffer for Data sieving)

- MPI-IO Hints can be given to improve performance by supplying more information to the library. This information can provide the link between application and file system.

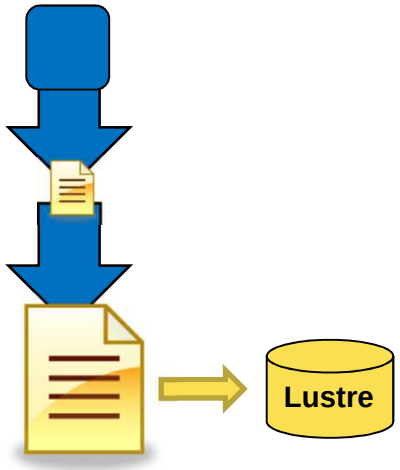
Buffered I/O

- Advantages
  - Aggregates smaller read/write operations into larger operations.
  - Examples: OS Kernel Buffer, MPI-IO Collective Buffering
- Disadvantages
  - Requires additional memory for the buffer.
  - Can tend to serialize I/O.
- Caution
  - Frequent buffer flushes can adversely affect performance.



Case Study: Buffered I/O

- A post processing application writes a 1GB file.
- This occurs from one writer, but occurs in many small write operations.
  - Takes 1080 s (~ 18 minutes) to complete.
- IO buffers were utilized to intercept these writes with 4 64 MB buffers.
  - Takes 4.5 s to complete. A 99.6% reduction in time.




```
File "ssef_cn_2008052600f000"
                    Calls          Seconds       Megabytes   Megabytes/sec   Avg Size
  Open                  1         0.001119
  Read                217         0.247026        0.105957        0.428931        512
  Write           2083634         1.453222     1017.398927      700.098632        512
  Close                 1         0.220755
  Total           2083853         1.922122     1017.504884      529.365466        512
  Sys Read              6         0.655251      384.000000      586.035160   67108864
  Sys Write            17         3.848807     1081.145508      280.904052   66686072
  Buffers used              4 (256 MB)
  Prefetches                6
  Preflushes               15
```


NICS

Outline

- Introduction to I/O
- Path from Application to File System
- Common I/O Considerations
- **I/O Best Practices**

I/O Best Practices

- **Read small, shared files from a single task**
  - Instead of reading a small file from every task, it is advisable to read the entire file from one task and broadcast the contents to all other tasks.
- **Small files (< 1 MB to 1 GB) accessed by a single process**
  - Set to a stripe count of 1.
- **Medium sized files (> 1 GB) accessed by a single process**
  - Set to utilize a stripe count of no more than 4.
- **Large files (>> 1 GB)**
  - set to a stripe count that would allow the file to be written to the Lustre file system.
  - The stripe count should be adjusted to a value larger than 4.
  - Such files should never be accessed by a serial I/O or file-per-process I/O pattern.

I/O Best Practices (2)

- Limit the number of files within a single directory
  - Incorporate additional directory structure
  - Set the Lustre stripe count of such directories which contain many small files to 1.

- Place small files on single OSTs
  - If only one process will read/write the file and the amount of data in the file is small (< 1 MB to 1 GB) , performance will be improved by limiting the file to a single OST on creation.

  → This can be done as shown below by: # lfs setstripe PathName -s 1m -i -1 -c 1

- Place directories containing many small files on single OSTs
  - If you are going to create many small files in a single directory, greater efficiency will be achieved if you have the directory default to 1 OST on creation

  →# lfs setstripe DirPathName -s 1m -i -1 -c 1

I/O Best Practices (3)

- Avoid opening and closing files frequently
  - Excessive overhead is created.
- Use ls -l only where absolutely necessary
  - Consider that “ls -l” must communicate with every OST that is assigned to a file being listed and this is done for every file listed; and so, is a very expensive operation. It also causes excessive overhead for other users. "ls" or "lfs find" are more efficient solutions.
- Consider available I/O middleware libraries
  - For large scale applications that are going to share large amounts of data, one way to improve performance is to use a middleware libary; such as ADIOS, HDF5, or MPI-IO.

Protecting your data HPSS

- The OLCF High Performance Storage System (HPSS) provides longer term storage for the large amounts of data created on the OLCF / NICS compute systems.
- The mass storage facility consists of tape and disk storage components, servers, and the HPSS software.
- Incoming data is written to disk, then later migrated to tape for long term archival.
- Tape storage is provided by robotic tape libraries

HPSS

- HSI
  - easy to use (FTP-like interface)
  - fine-grained control of parameters
  - works well for small numbers of large files
- HTAR
  - works like tar command
  - treats all files in the transfer as one file in HPSS
  - preferred way to handle large number of small files
- More information on NICS/OLCF website
  - http://www.nics.tennessee.edu/computing-resources/hpss
  - http://www.olcf.ornl.gov/kb_articles/hpss/

Further Information

- NICS website
  - http://www.nics.tennessee.edu/I-O-Best-Practices
- Lustre Operations Manual
  - http://dlc.sun.com/pdf/821-0035-11/821-0035-11.pdf
- The NetCDF Tutorial
  - http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-tutorial.pdf
- Introduction to HDF5
  - http:// ww.hdfgroup.org/HDF5/doc/H5.intro.html

Further Information MPI-IO

– Rajeev Thakur, William Gropp, and Ewing Lusk, "A Case for Using MPI's Derived Datatypes to Improve I/O Performance," in *Proc. of SC98: High Performance Networking and Computing*, November 1998.
  - http://www.mcs.anl.gov/~thakur/dtype

– Rajeev Thakur, William Gropp, and Ewing Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182-189.
  - http://www.mcs.anl.gov/~thakur/papers/romio-coll.pdf

– Getting Started on MPI I/O, Cray Doc S–2490–40, December 2009.
  - http://docs.cray.com/books/S-2490-40/S-2490-40.pdf

Thank You !