
# Parallel I/O Techniques and Performance Optimization

Lonnie Crosby
lcrosby1@utk.edu

NICS Scientific Computing Group

NICS/RDAV Spring Training 2012

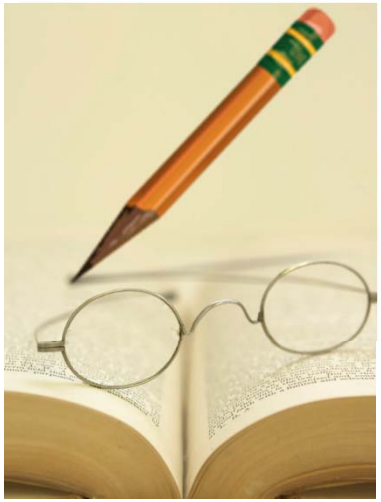
March 22, 2012

# Outline

- Introduction to I/O
- Path from Application to File System
  - Data and Performance
  - I/O Patterns
  - Lustre File System
  - I/O Performance Results
- MPI-IO
  - General File I/O
  - Derived MPI DataTypes
  - Collective I/O
- Common I/O Considerations
- Application Examples

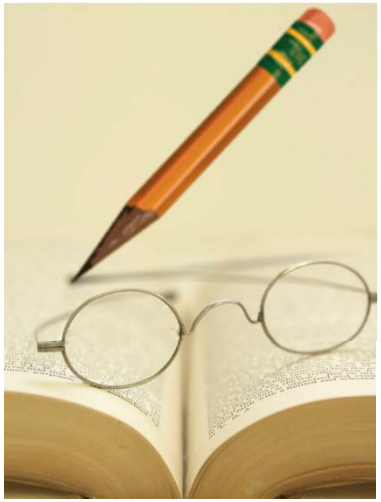
# Factors which affect I/O.

- 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).





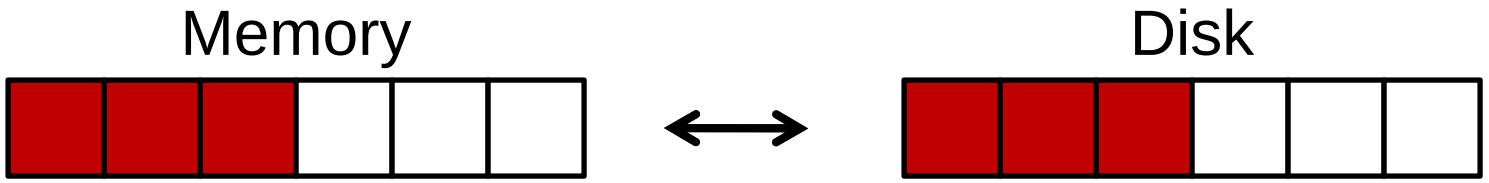
NICS

# Outline

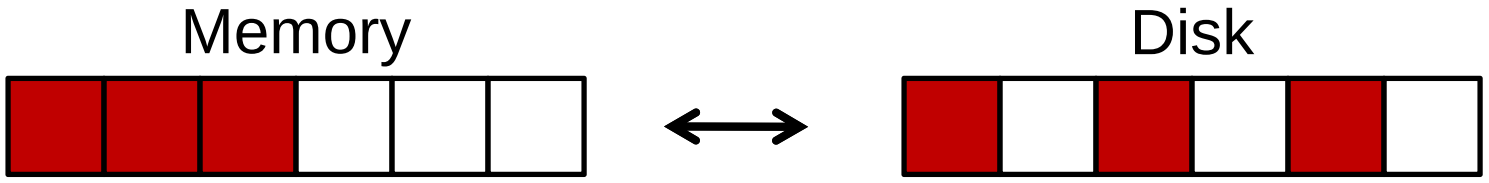
- Path from Application to File System
  - Data and Performance
  - I/O Patterns
  - Lustre File System
  - I/O Performance Results

# Data and Performance

- The best performance comes from situations when the data is accessed contiguously in memory and on disk.

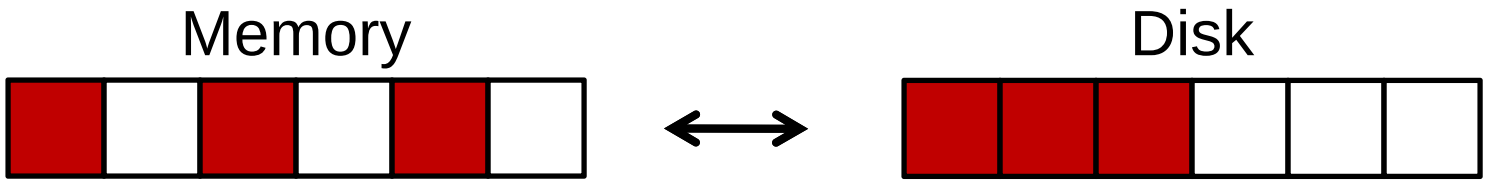



- Commonly, data access is contiguous in memory but noncontiguous on disk. For example, to reconstruct a global data structure via parallel I/O.

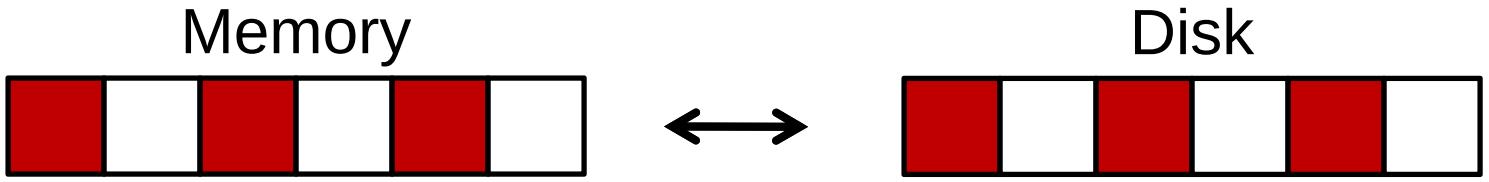


# Data and Performance

- Sometimes, data access may be contiguous on disk but noncontiguous in memory. For example, writing out the interior of a domain without ghost cells.

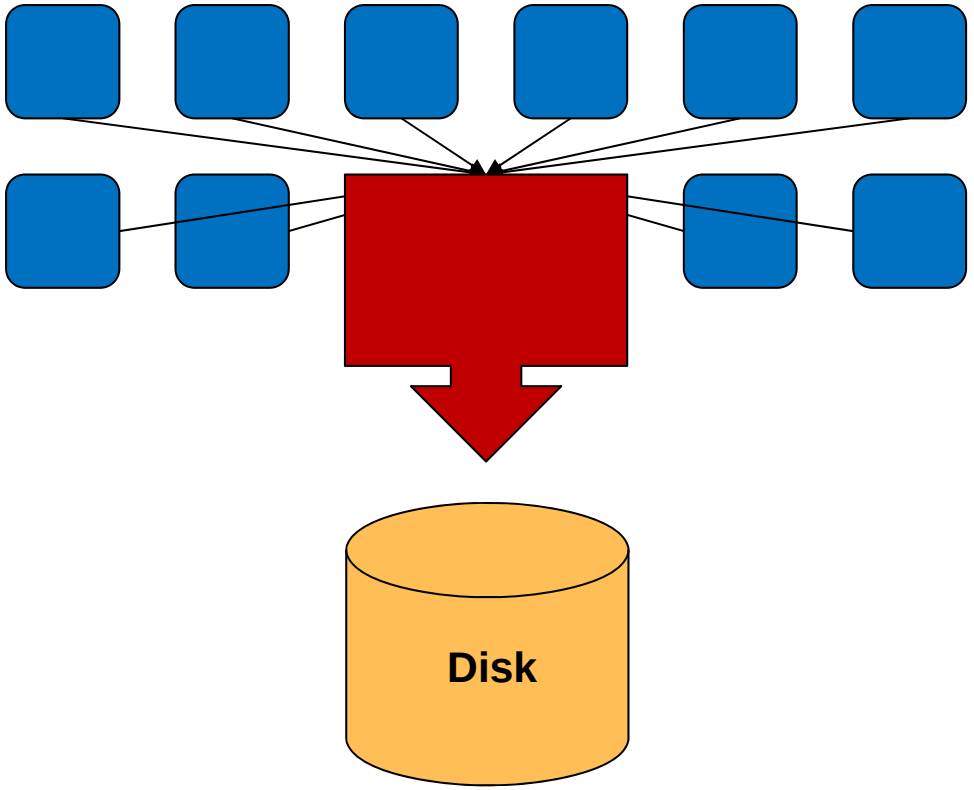



- A large impact on I/O performance would be observed if data access was noncontiguous both in memory and on disk.



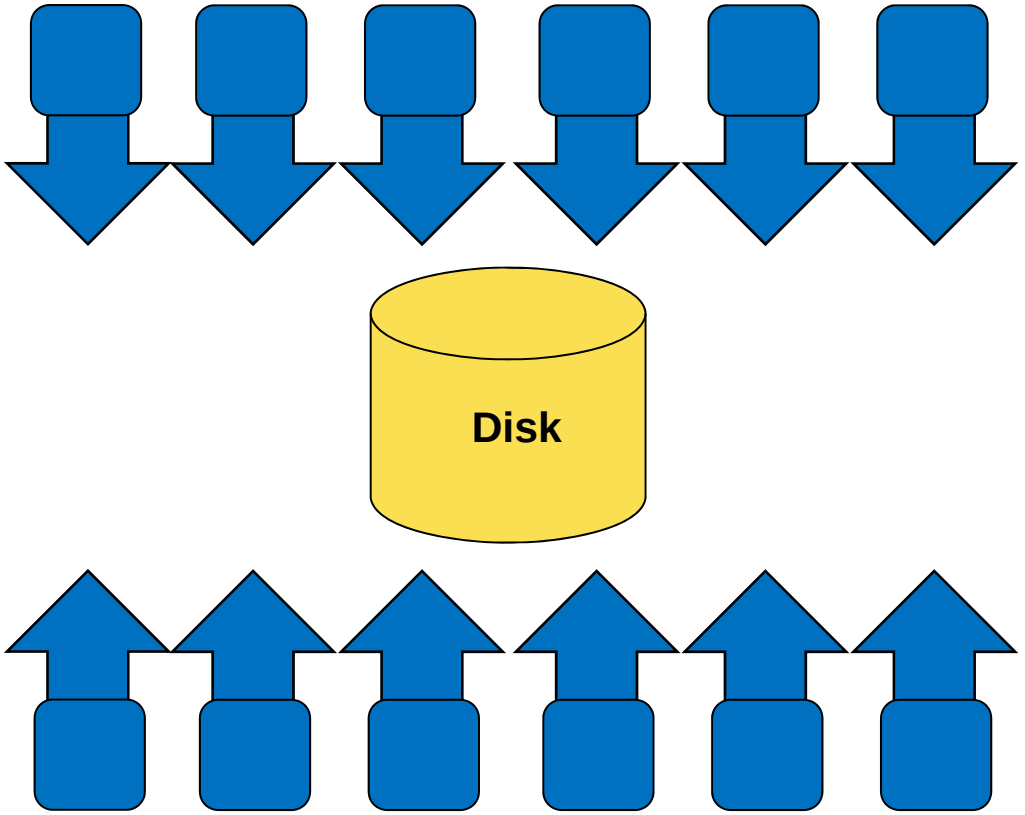
# Serial I/O: Spokesperson

- Spokesperson
  - One process performs I/O.
    - Data Aggregation or Duplication (I/O size large)
    - Performance limited by single I/O process.
  - Pattern does not scale.
    - I/O rate does not increase with process count.



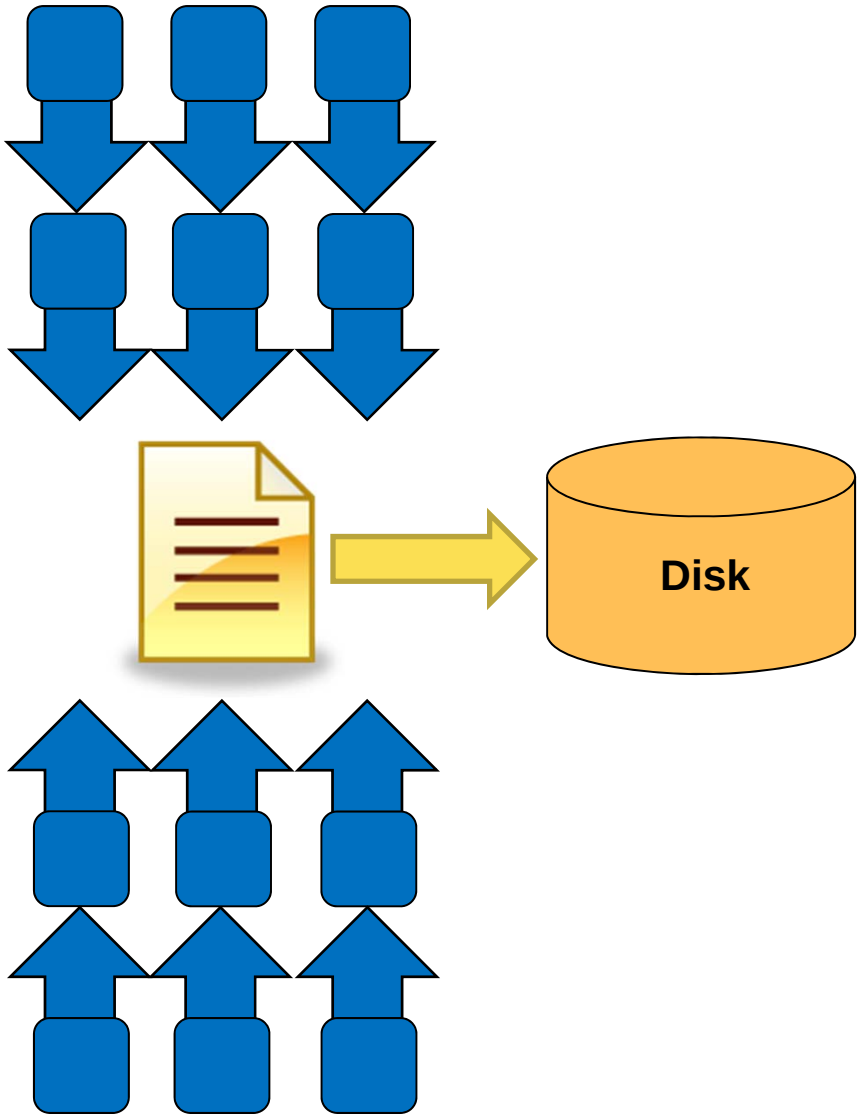
# Parallel I/O: File-per-Process

- File per process
  - All processes perform I/O to individual files.
    - I/O rate increases with number of files.
    - Limited by file system.
    - I/O size small
  - Pattern does not scale at large process counts.
    - Hardware constraints such as number of disks, arrays, etc. limits the I/O rate increase with number of processes.



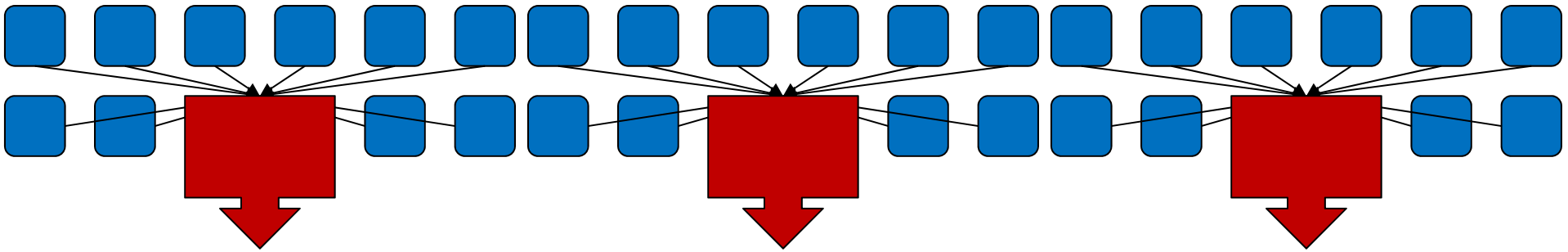
# Parallel I/O: Shared File

- Shared File
  - Each process performs I/O to a single file which is shared.
    - I/O rate increases with number of processes
    - Limited by filesystem
    - I/O size small
  - Performance
    - Data layout within the shared file is very important.



# Pattern Combinations

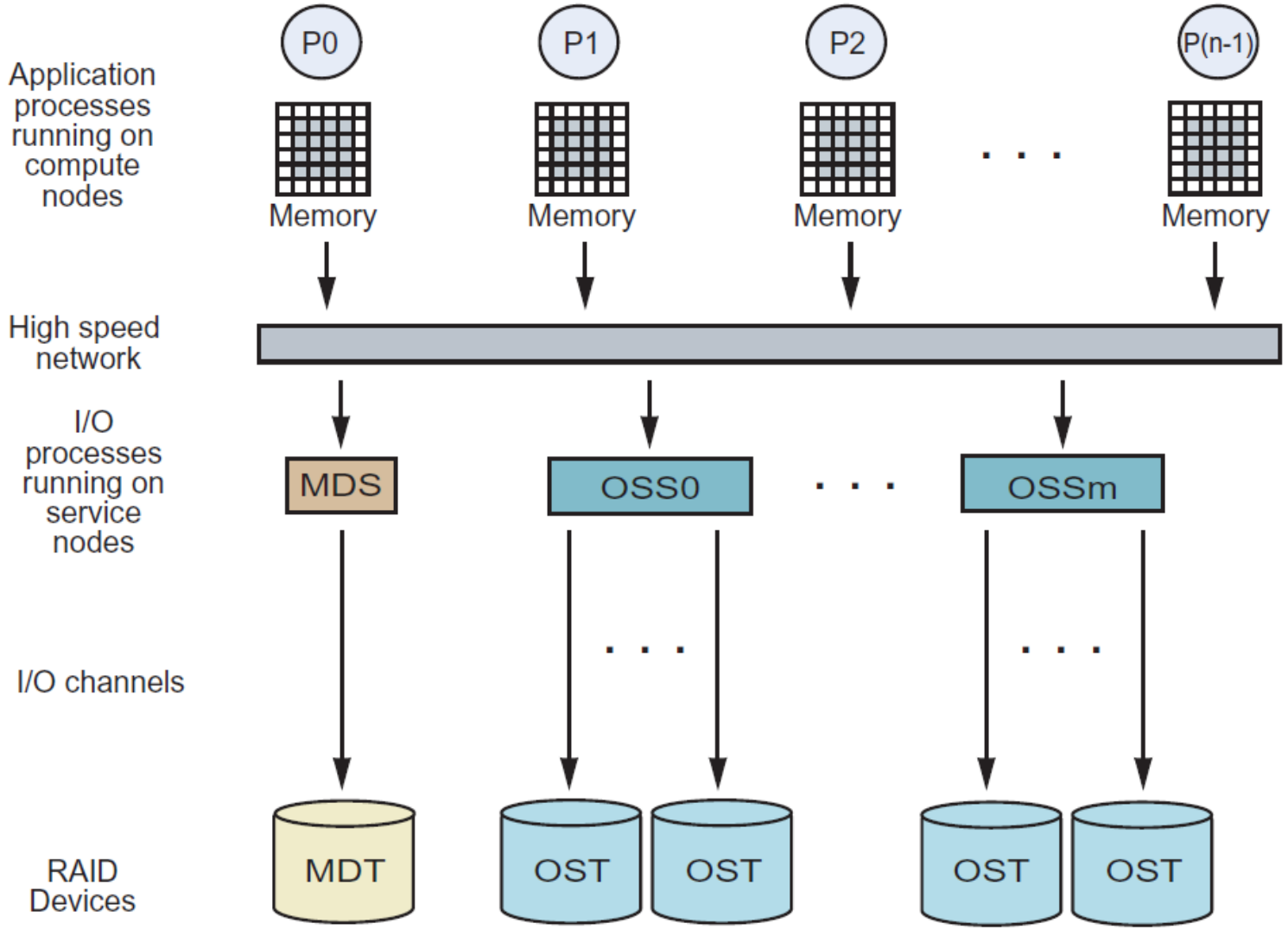
- Subset of processes which perform I/O.
  - Aggregation of a group of processes data.
    - Serializes I/O in group. Increases I/O size.
  - I/O process may access independent or shared files.
    - Limits the number of files accessed.
  - Group of processes perform parallel I/O to a shared file.
    - Increases the number of shared files to increase file system usage.
    - Decreases number of processes which access a shared file to 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.

# A Bigger Picture: Lustre File System



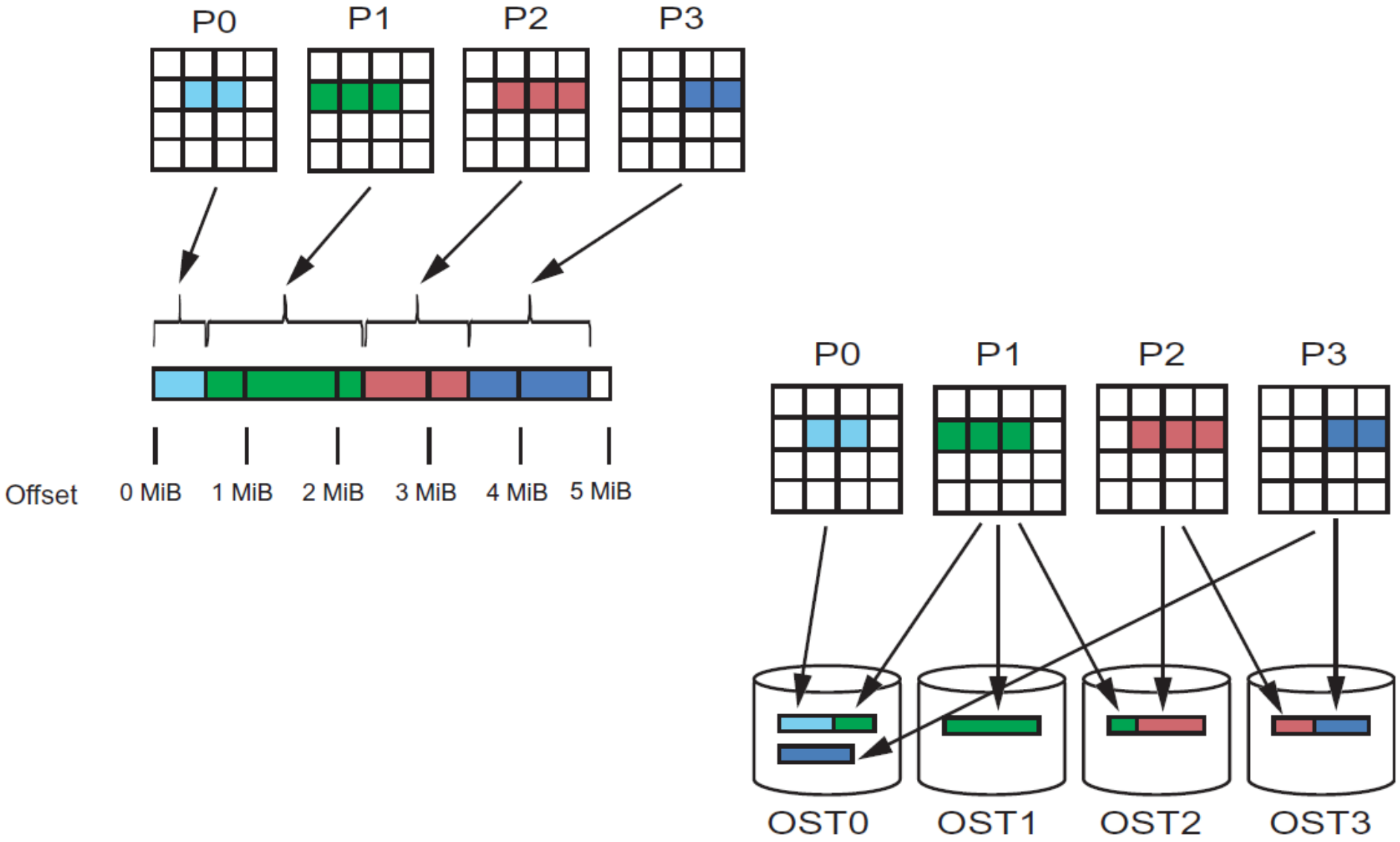
# Striping on the Lustre File system

- lfs setstripe and getstripe command
  - Syntax:  lfs setsripe –c <stripe_count> -s <stripe_size> -i <stripe_index> <file or directory>
  - <stripe_count>
    - 0 (Use default)
    - -1 (Use all available OSTs, max = 160)
  - <stripe_size>
    - 0 (Use default)
    - In bytes, although can be specified with k, m or g (in KB, MB and GB respectively)
  - <stripe_index>
    - -1 (allow MDS to choose starting OST, recommended)
  - <file or directory>
    - Cannot change the striping characteristics of existing files
    - Striping characteristics of directories can be changed at any time.

# LFS setstripe and getstripe commands

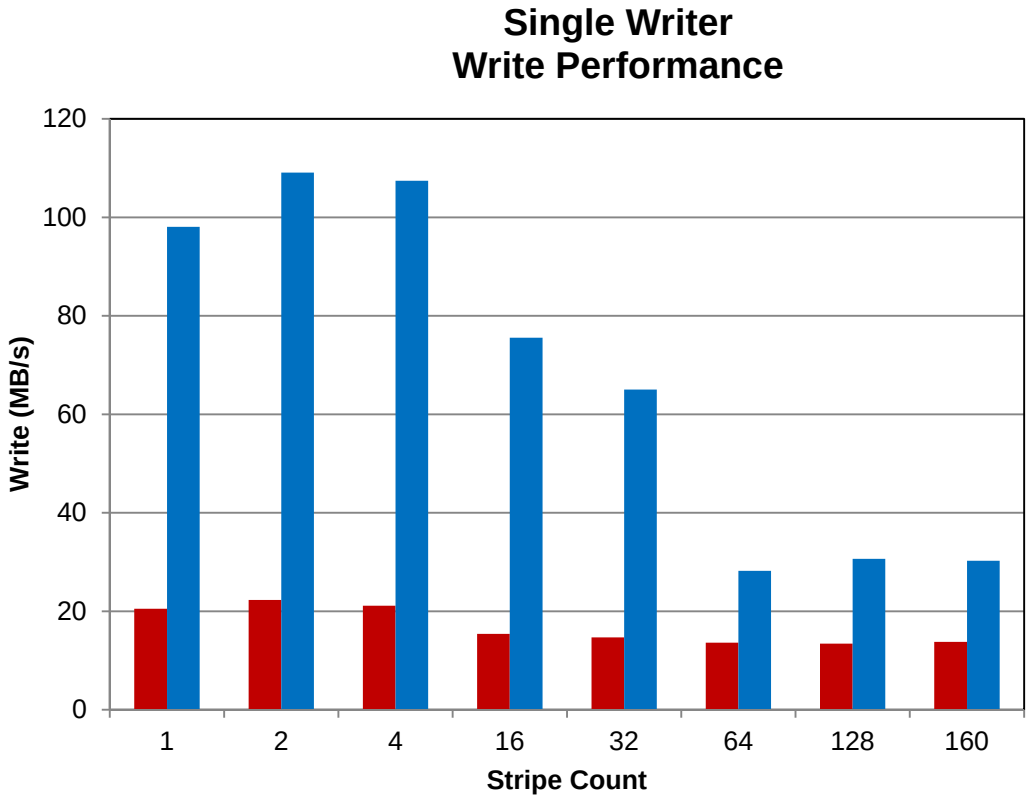
- Example:
  - % lfs setstripe -c 5 -s 2M -i 27 test_file_stripe
  - % lfs getstripe test_file_stripe
  - test_file_stripe
  - lmm_stripe_count:  5
  - lmm_stripe_size:  2097152
  - lmm_stripe_offset:  27
  - obdidx objid objid group
  - 27 29421438 0x1c0ef7e 0
  - 97 29211011 0x1bdb983 0
  - 87 29386728 0x1c067e8 0
  - 90 28982042 0x1ba3b1a 0
  - 13 29013598 0x1bab65e 0

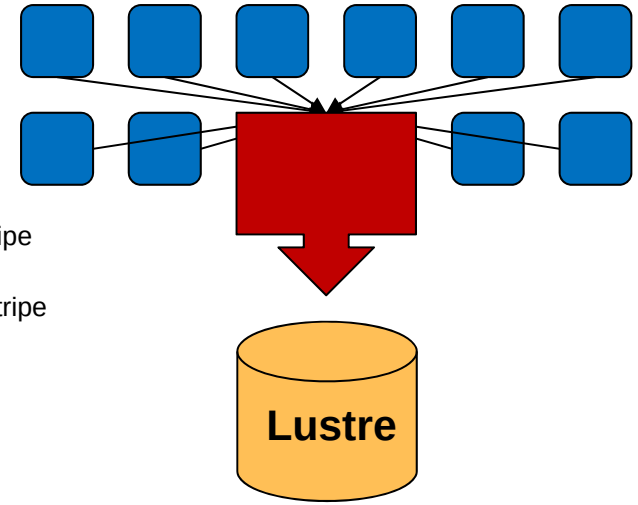
# File Striping: Physical and Logical Views



# Single writer performance and Lustre

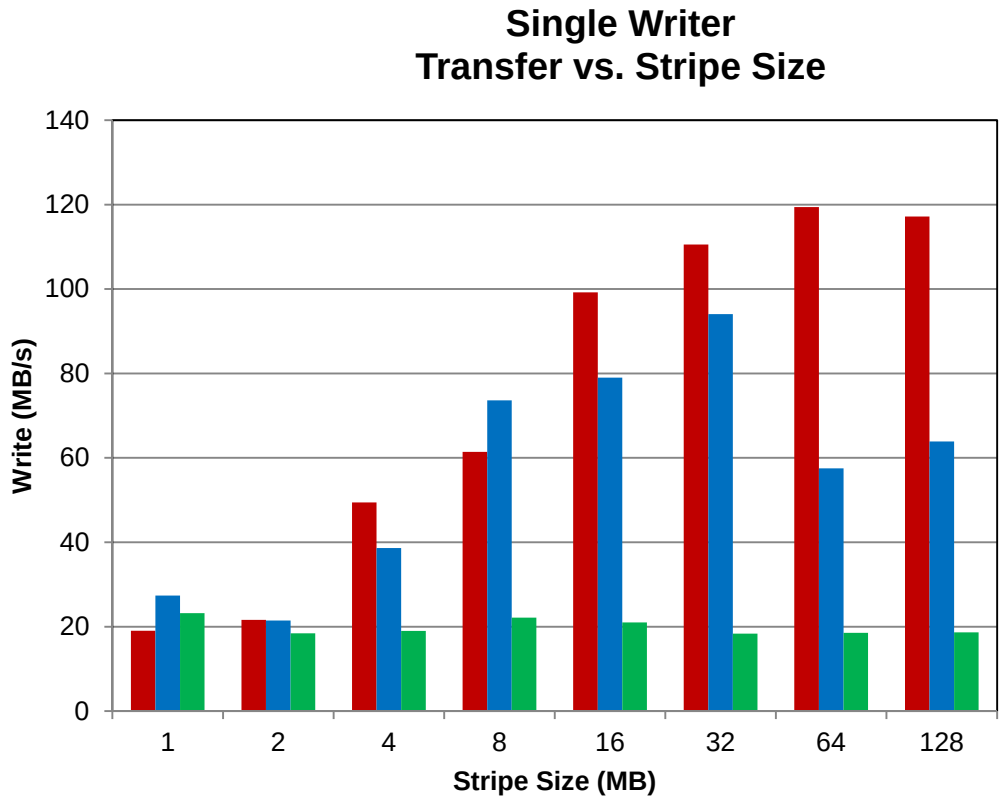
- File size 32 MB – 5 GB (32 MB per OST)
- 32 MB I/O write (transfer) size (large)
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance

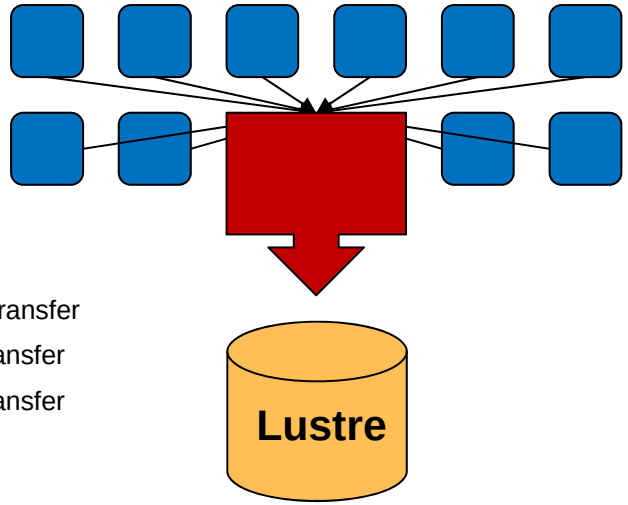


# Stripe size and I/O Operation size

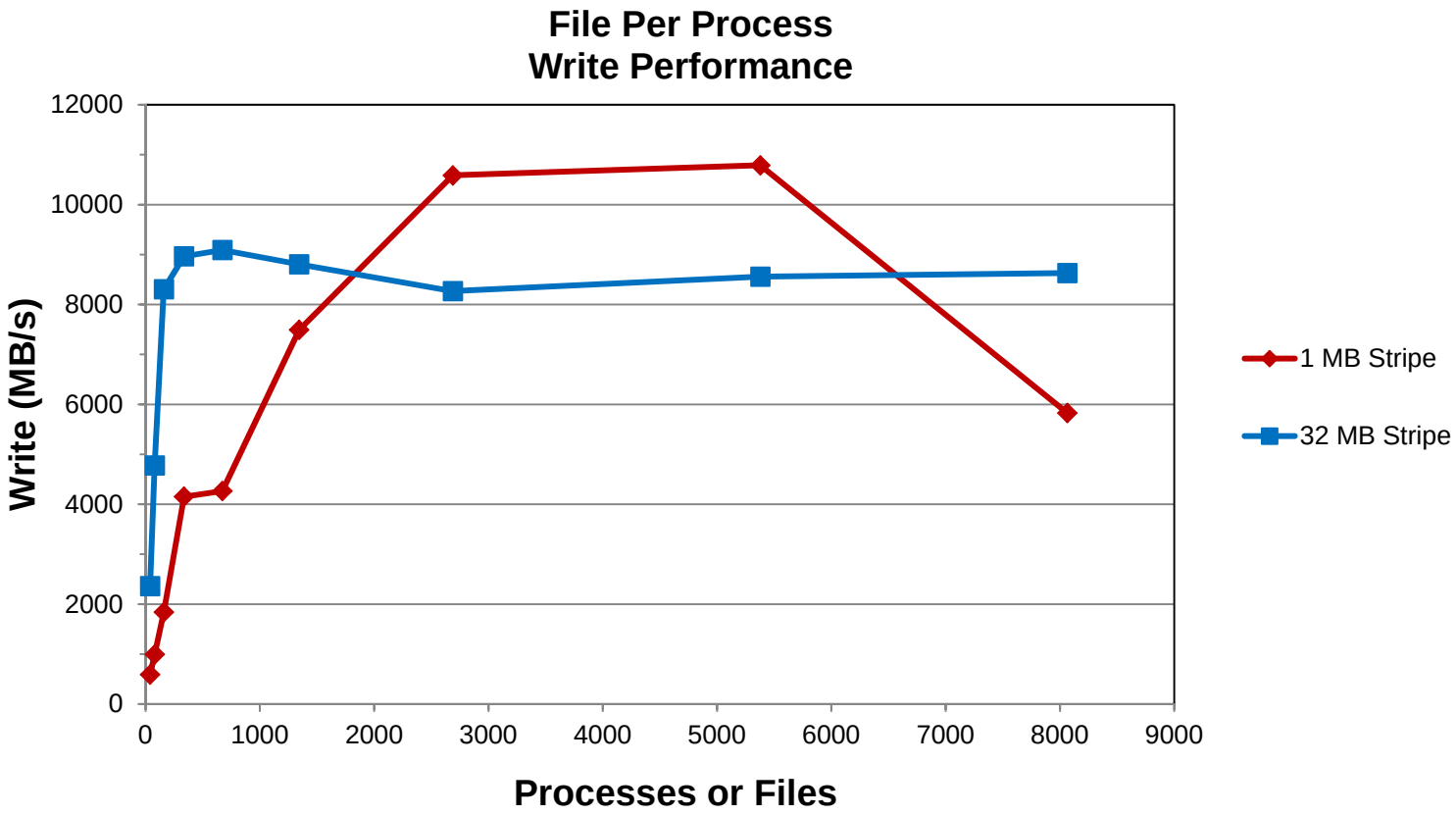
- Single OST, 256 MB File Size
  - Performance can be limited by the process (transfer size) or file system (stripe size)



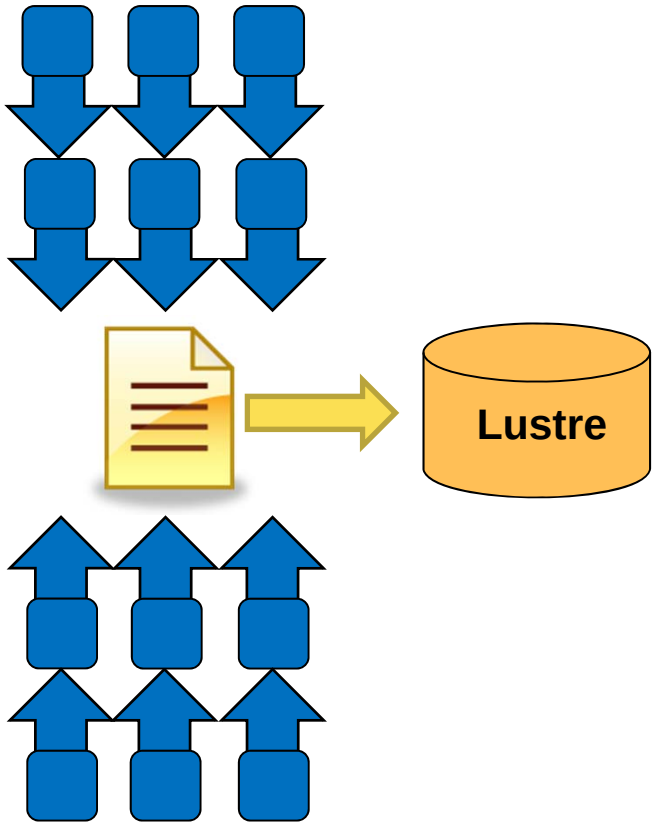



# Scalability:  File Per Process

- 128 MB per file and a 32 MB Transfer size



# Single Shared Files and Lustre Stripes



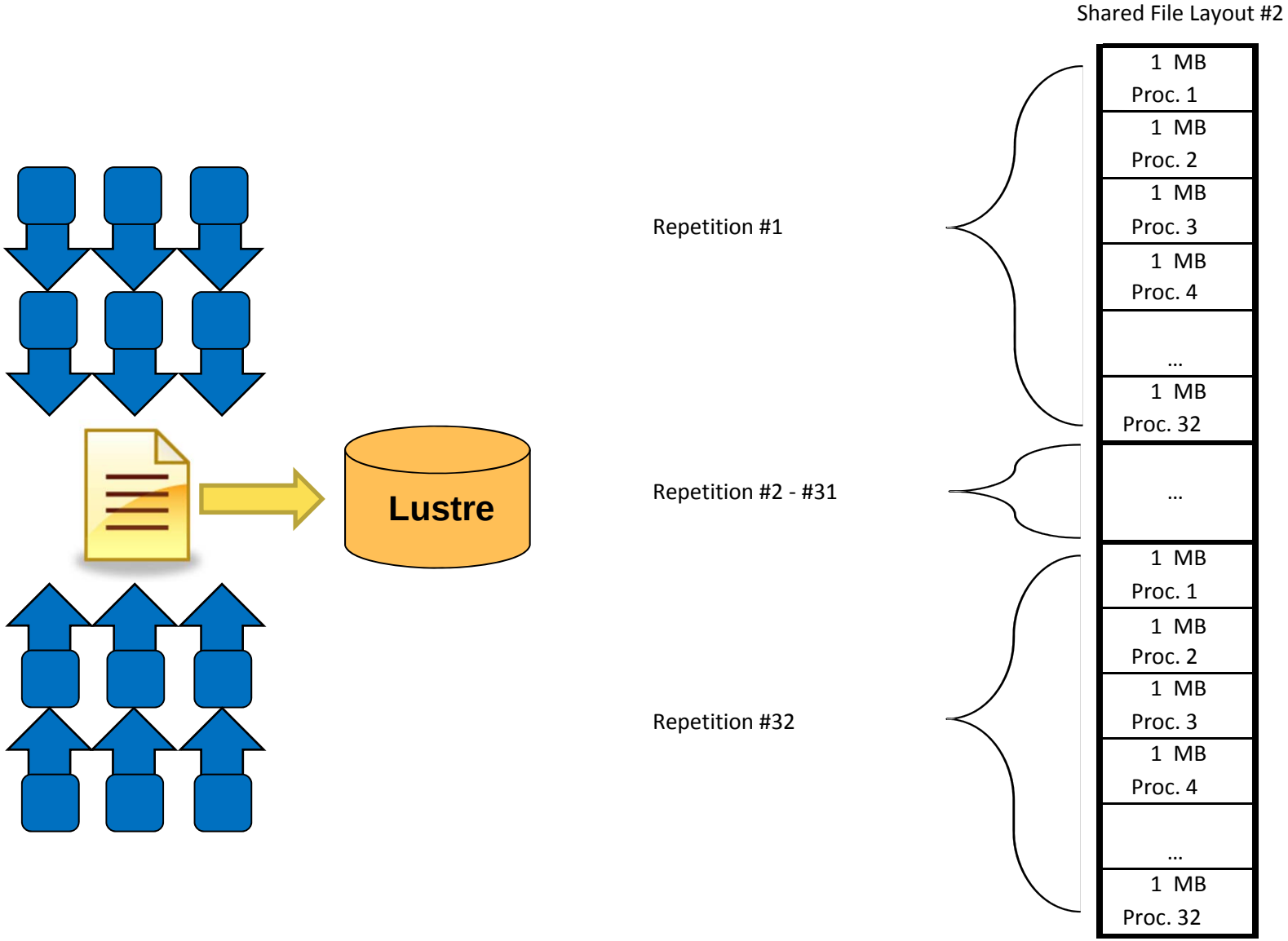

Shared File Layout #1

| |
|---|
| 32 MB<br>Proc. 1 |
| 32 MB<br>Proc. 2 |
| 32 MB<br>Proc. 3 |
| 32 MB<br>Proc. 4 |
| … |
| 32 MB<br>Proc. 32 |

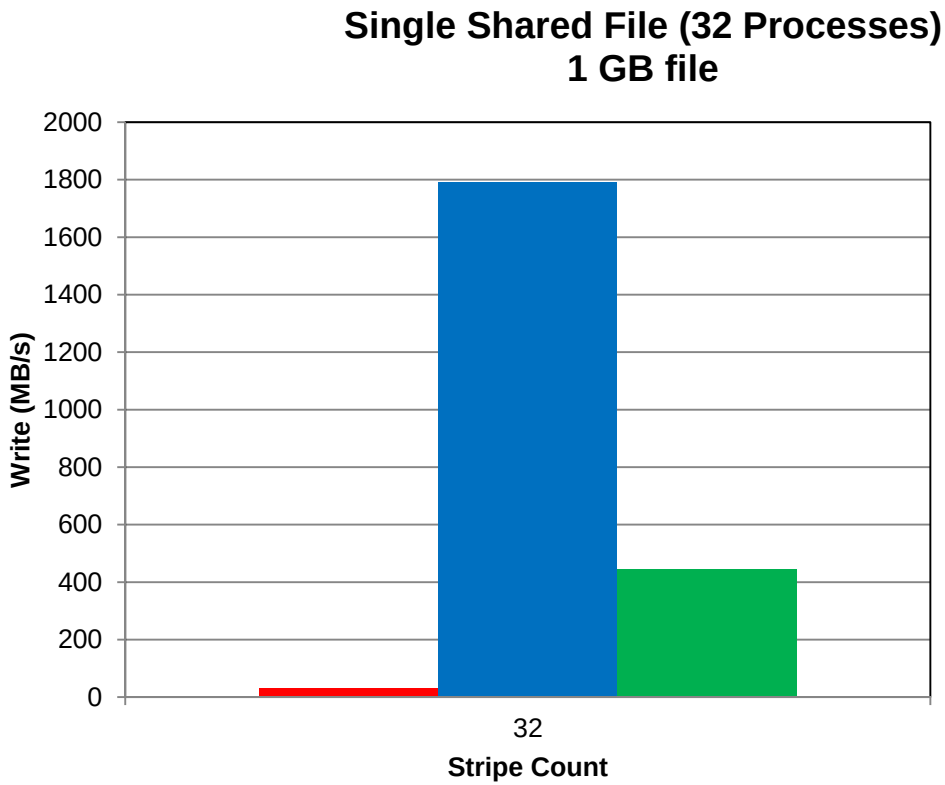


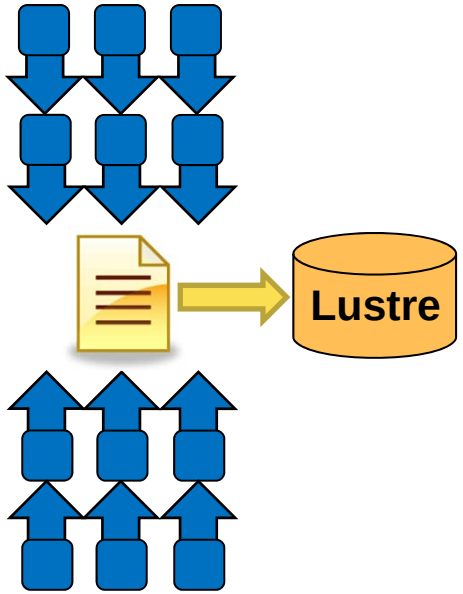
NICS

# Single Shared Files and Lustre Stripes



# File Layout and Lustre Stripe Pattern



# Summary

- 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).
- 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 by the file system.
  - Potential solution is to utilize multiple shared files or a subset of processes which perform I/O.

# Outline

- MPI-IO
  - General File I/O
  - Derived MPI DataTypes
  - Collective I/O

# I/O Libraries (MPI-IO)

- Many I/O libraries such as HDF5 and Parallel NetCDF are built on top of MPI-IO.
- Such libraries are abstractions from MPI-IO.

- Such implementations allow for higher information propagation to MPI-IO (without user intervention).
- Understand information flow through MPI-IO and how this may affect performance.

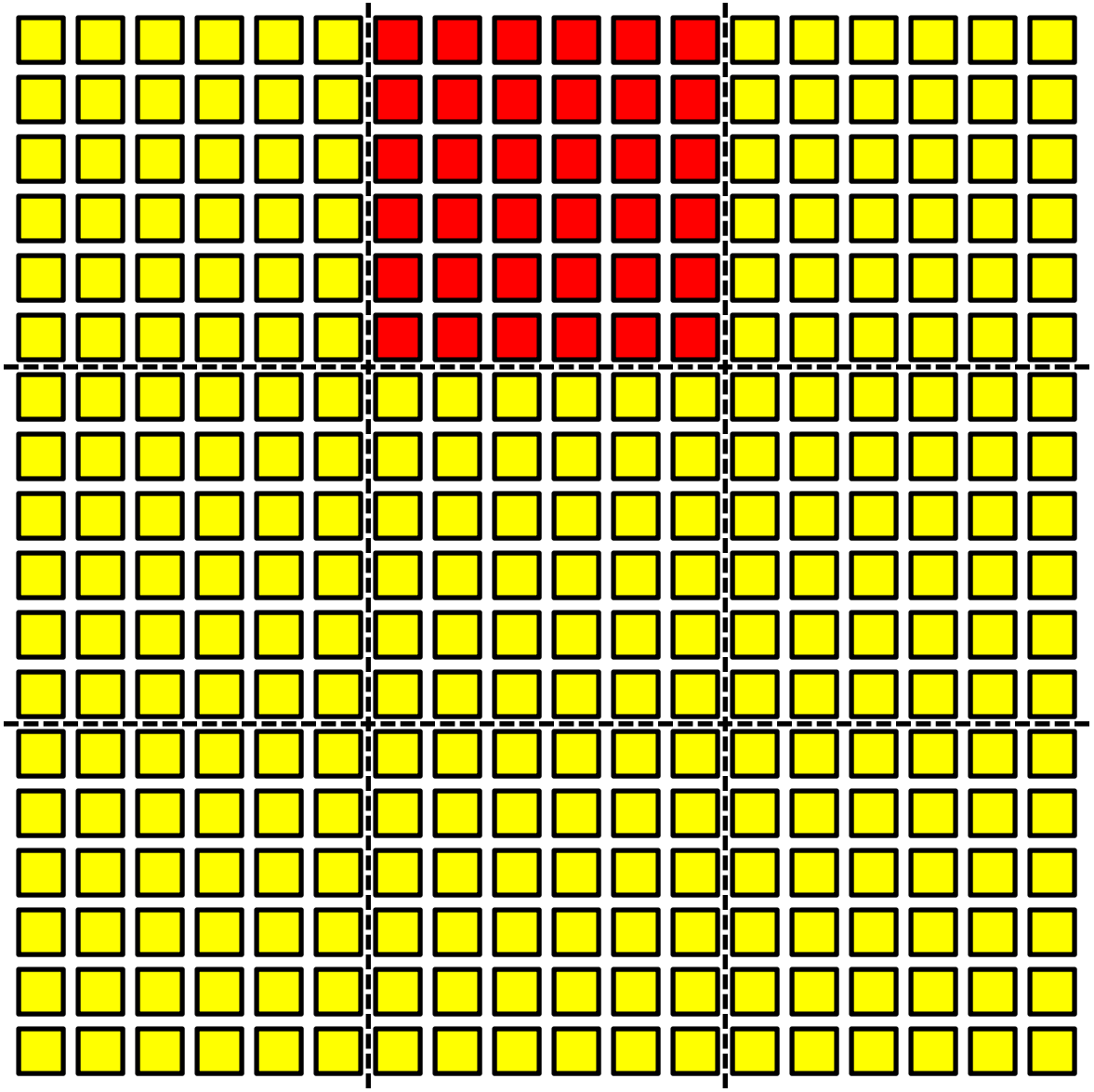
# MPI I/O: Opening a File

- int MPI_File_open ( MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh )
  - Fortran: Subroutine with additional argument (integer ierr). MPI_File, MPI_Info, and MPI_Comm data types are integers in Fortran.
  - File is opened for each member of MPI_comm comm. MPI_COMM_SELF may be used for a private file.
  - int amode allows the file to be opened Read or Write only.
  - MPI_INFO_NULL may be used for MPI_Info info. May set hints specific to this file. See MPICH_MPIIO_HINTS.

# MPI Derived Data Types

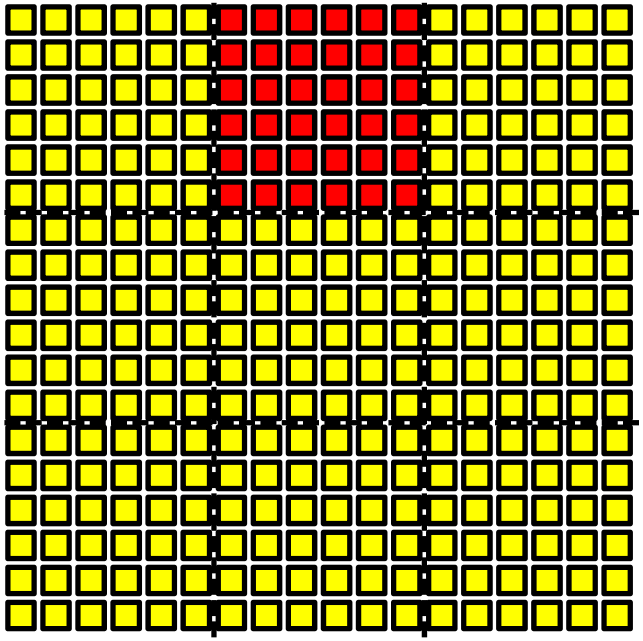
- User defined data types which are made up of elementary data types such as MPI_DOUBLE or MPI_INTEGER.
- Derived data types can contain “holes” which are used to read or write noncontiguous data.
- Derived data types pass information to the MPIIO implementation  which allows for better performance.

# Subarray Data Type



- Parameters
  - Global (18 x 18)
  - Subarray (6 x 6)
  - Index = {0, 6}
  - Extent of data type is 324 elements.
- Subarray contains the data. Remaining portions of the global array are “holes”.
- Must define how global array is laid out in memory (column or row major, i.e. Fortran or C)

# Subarray Data Type (Linearized)



- Column Major (Fortran Ordering)


- Row Major (C Ordering)




NICS

# Vector Data Type


- Parameters
  - 6 Blocks (One for each row or column, are contiguous)
  - Blocksize = 6 elements
  - Stride = 18 (Elements between the beginning of each block)
  - Extent of data type is 96 elements.
- Blocks contain data
- Elements not within blocks are “holes” in the data type.

# MPI Data type syntax

- int MPI_Type_vector ( int count, int blocklen, int stride, MPI_DataType oldtype, MPI_Datatype *newtype )
- int MPI_Type_create_subarray ( int ndims, int *array_of_sizes, int *array_of_subsizes, int *array_of_starts, int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Fortran: These are subroutines with an additional argument at the end (integer ierr). The MPI_Datatype C data types are integers in Fortran.
  - Data types must be committed before use via:
    - int MPI_Type_commit ( MPI_Datatype *datatype )

# Describing the file: MPI_File_set_view

- int MPI_File_set_view ( MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info_info )
  - Fortran: Subroutine with additional argument (integer ierr). MPI_File, MPI_Info, MPI_Offset, and MPI_Datatype data types are integers in Fortran.
  - etype is a data type which forms the basis of file access. Offset is in terms of etype.
  - Filetype is a data type which describes the portions of the file for which data will be written.
  - datarep may be 'NATIVE' for machine dependent binary.
  - MPI_INFO_NULL may be used for MPI_Info info. May set hints specific to this file. See MPICH_MPIIO_HINTS.

# Information flow through MPI-IO

- Three Levels of I/O possible within MPI-IO.
  - Explicit Read/Write (No use of derived datatypes)
  - Use of MPI Derived Data types (non-contiguous data)
  - Collective I/O (parallel I/O to a shared file)

# Information in file reads/writes.

- Explicit Read/Write


  - MPI_File_set_view (Offset = 108)
  - MPI_File_write (6 elements)
  - MPI_File_seek (12 elements)
  - Repeat (6 times)
  - MPI_File_write_at (Uses explicit offsets, combines write and seek)

# Information in file reads/writes.


- Using Derived Data Types
  - MPI_Type_vector


  - MPI_Type_create_subarray


  - MPI_File_set_view (Offset = 108 or Offset = 0, filetype = vector or filetype = subarray)
  - MPI_File_write_at (36 elements )



NICS

# Collective I/O

- The use of MPI_File_write [read]_at_all or MPI_File_write [read]_all allows for collective I/O using shared file pointers.
- Information can be given to MPI-IO via MPI derived data types. However, additional information can be given to MPI-IO (between MPI ranks) by using collective I/O.
- Minimizes the number of independent file accesses. Additionally allows collective mechanisms such as collective buffering and data sieving to be used.

# Read/Write Syntax

- int MPI_File_write [read]_at_all ( MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status )
  - Fortran: These are subroutines with an additional argument at the end (integer ierr). The MPI_Datatype, MPI_Offset, and MPI_Status C data types are integers in Fortran.
  - Difference between MPI_File_write [read] is the MPI_Offset offset argument. MPI_File_write [read]_at has the same arguments.
  - MPI_STATUS_IGNORE can be used for MPI_Status *status

# Closing Files and Freeing Memory

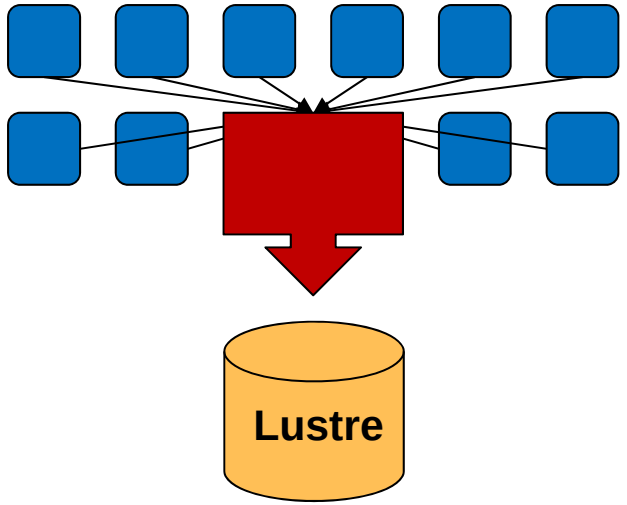
- int MPI_File_close ( MPI_File *fh )
- int MPI_Type_free ( MPI_Datatype *datatype )
  - Fortran: These are subroutines with an additional argument at the end (integer ierr). The MPI_Datatype and MPI_File C data types are integers in Fortran.

# 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 )
- export MPICH_MPIIO_HINTS = ‘ pathname pattern : key=value : key2=value2 : …’

# Collective Buffering and Data Sieving

- Collective Buffering
  - Aggregates I/O to a process (buffer)
  - This buffer is then written to disk.



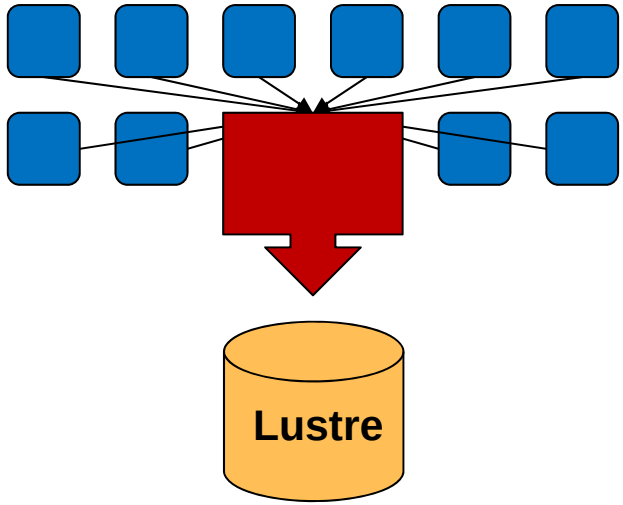

- Data Sieving
  - More data than needed is written/read (buffer).
  - The needed information is obtained from the buffer.

# Common I/O Considerations

- Standard Input/Output
- Buffered I/O
- Binary Files and Endianess
- Subsetting I/O
  - Aggregation
  - Turnstile
  - Multiple Shared Files

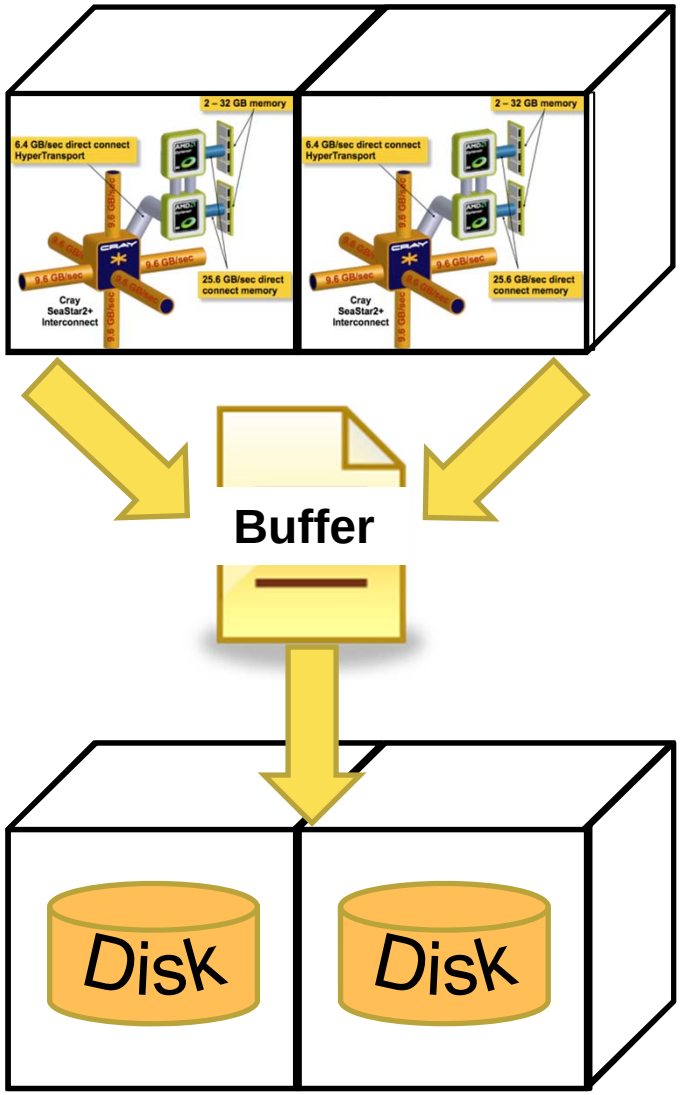
# Standard Output and Error

- Standard Output and Error streams are effectively serial I/O.
- Generally, the MPI launcher will aggregate these requests. (Example:  mpirun, mpiexec, aprun, ibrun, etc..)
- Disable debugging messages when running in production mode.
  - "Hello, I'm task 32000!"
  - "Task 64000, made it through loop."



# 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.

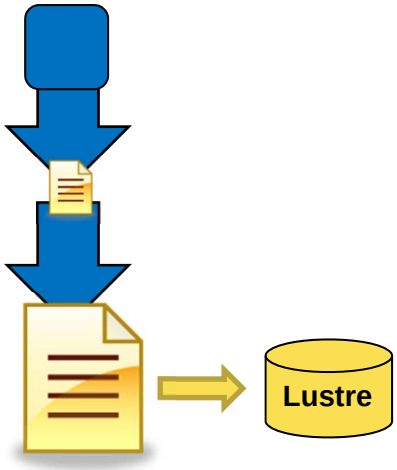


# IOBuf Library

- An easy to use, tunable I/O buffering library which may be helpful to both improve I/O performance (in some circumstances) and gather I/O statistics.
- Use
  - % module load iobuf
  - Use the Cray wrappers (ftn, cc, CC) to relink your application.
  - % export IOBUF_PARAMS='*:verbose' (To obtain summary information when file is closed)
- Other settings can be utilized through the IOBUF_PARAMS environmental variable (see: % man iobuf for more information)

# 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

# Binary Files and Endianess

- **Writing a big-endian binary file with compiler flag byteswapio**

```
File "XXXXXX"
                    Calls        Megabytes      Avg Size
  Open                  1
  Write           5918150      23071.28062          4088
  Close                 1
  Total           5918152      23071.28062          4088
```

- **Writing a little-endian binary**

```
File "XXXXXX"
                    Calls        Megabytes      Avg Size
  Open                  1
  Write               350      23071.28062      69120000
  Close                 1
  Total               352      23071.28062      69120000
```



NICS

# Subsetting I/O

- At large core counts, I/O performance can be hindered
  - by the collection of metadata operations (File-per-process) or
  - by file system contention (Single-shared-file).
- One solution is to use a subset of application processes to perform I/O. This limits
  - the number of files (File-per-process) or
  - the number of processes accessing file system resources (Single-shared-file).
- If you can not implement a subsetting approach, try to limit the number of synchronous file opens to reduce the number of requests simultaneously hitting the metadata server.

# Application Examples

Crosby, L. D.; Brook, R. G.; Rekepalli, B.; Sekachev, M.; Vose, A.; Wong, K. "A Pragmatic Approach to Improving the Large-scale I/O Performance of Scientific Applications." *Cray User Group Inc.: Conference Proceedings*, 2011

.PICMSS (The Parallel Interoperable Computational Mechanics Simulation System)

- A computational fluid dynamics (CFD) code used to provide solutions to incompressible problems. Developed at the University of Tennessee's CFD laboratory.

- AWP-ODC (Anelastic Wave Propagation)
  - Seismic code used to conduct the "M8" simulation, which models a magnitude 8.0 earthquake on the southern San Andreas fault. Development coordinated by Southern California Earthquake Center (SCEC) at the University of Southern California.

- BLAST (Basic Local Alignment Search Tool)
  - A parallel implementation developed at the University of Tennessee, capable of utilizing 100 thousand compute cores.

# Application #1

- Computational Grid
  - 10,125 x 5,000 x 1,060 global grid nodes (5,062 x 2,500 x 530 effective grid nodes)
  - Decomposed among 30,000 processes via a process grid of 75 x 40 x 10 processes. (68 x 63 x 53 local grid nodes)
  - Each grid stored column-major.
- Application data
  - Three variables are stored per grid point in three arrays, one per variable (local grid). Multiple time steps are stored by concatenation.
- Output data
  - Three shared files are written, one per variable, with data ordered corresponding to the global grid. Multiple time steps are stored by concatenation.

# Optimization

## Original Implementation

- Derived Data type created via MPI_Type_create_hindexed
  - Each block consists of a single value placed by an explicit offset.

## Optimized Implementation

- Derived Data type created via MPI_Type_create_subarray
  - Each block consists of a contiguous set of values (column) placed by an offset.

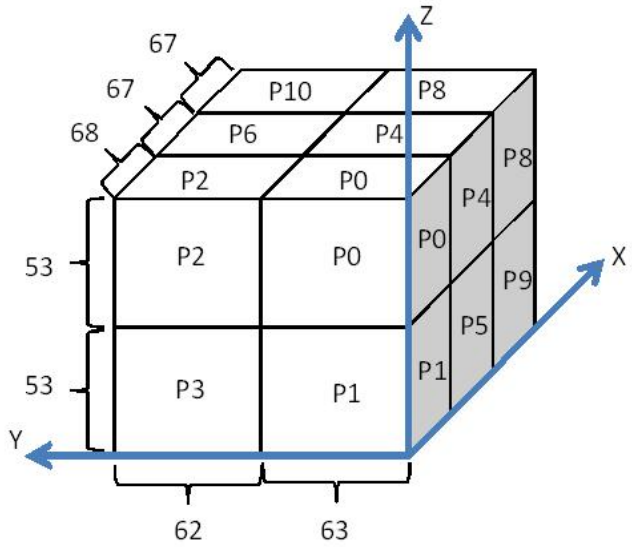



**Figure 1:** The domain decomposition of a 202x125x106 grid among 12 processes in a 3x2x2 grid. The process assignments are listed P0-P11 and the numbers in brackets detail the number of grid nodes along each direction for each process block.

# Results

- Collective MPI-IO calls are utilized along with appropriate Collective-buffering and Lustre stripe settings.
  - Stripe count = 160  Stripe Size = 1MB

- Given amount of data
  - Optimization saves 12 min/write
  - Over 200 time steps, savings of about 2 hours.

Table 1: Comparison between Original and Optimized I/O in Application 1

| | | Bandwidth (GB/s) | |
|---|---|---|---|
| Time Step | Data (TB) | Original | Optimized |
| 20 | 1.46 | 1.05 | 2.03 |
| 40 | 1.46 | 1.06 | 1.66 |
| 60 | 1.46 | 1.40 | 2.16 |
| 80 | 1.46 | 1.16 | 1.94 |
| 100 | 1.46 | 1.27 | 2.30 |
| 120 | 1.46 | 1.55 | 2.06 |
| 140 | 1.46 | 0.98 | 1.70 |
| 160 | 1.46 | 1.30 | 2.65 |
| 180 | 1.46 | 0.87 | 3.34 |
| 200 | 1.46 | 1.13 | 2.99 |

# Application #2

- Task based parallelism
  - Hierarchical application and node-level master processes who serve tasks to node-level worker processes
  - Work is obtained by worker processes via a node-level master process. The application-level master provides work to the node-level master processes.
  - I/O is performed per node via a dedicated writer process.
- Application data
  - Each worker produces XML output per task. These are concatenated by the writer process and compressed.
- Output data
  - A file per node is written which consists of a concatenation of compressed blocks.

# Optimization

## Original Implementation

- On-demand compression and write.
  - When the writer process receives output from a worker it is immediately compressed and written to disk.
- Implications
  - Output files consist of a large number of compressed blocks each with a 4-byte header.
  - Output files written in a large number of small writes.

## Optimized Implementation

- Dual Buffering
  - A buffer for uncompressed XML data is created. Once filled, the concatenated data is compressed.
  - A buffer for compressed XML data is created. Once filled, the data is written to disk.
- Implications
  - Output files consist of a few, large compressed blocks each with a 4-byte header.
  - Output files written in a few, large writes.

# Results

- Benchmark case utilizes 24,576 compute cores (2,048 nodes) Optimized case utilizes 768 MB buffers.
  - Stripe count = 1  Stripe Size = 1MB

- Compression Efficiency
  - Compression ratio of about 1:7.5
  - Compression takes longer than the file write.
  - With optimizations, file write would take about 2.25 seconds without prior compression.

Table 2: Comparison between Original and Optimized I/O in Application 2

| | Original | Optimized |
|---|---|---|
| Average Compression Time (s) | 11.93 | 8.85 |
| Std. Dev. (s) | 0.79 | 0.46 |
| Bandwidth (MB/s) | 25.75 | 34.63 |
| Average Write Time (s) | 10.47 | 0.30 |
| Std. Dev. (s) | 8.36 | 1.06 |
| Bandwidth (MB/s) | 16.04 | 466.67 |

# Application #3

- Computational Grid
  - $256^3$ global grid nodes
  - Decomposed among 3,000 processes via XY slabs in units of X columns. The local grid corresponds slabs of about 256 x 22 nodes.
  - Six variables per grid node is stored.
  - Each grid stored column-major.
- Application data
  - A column-major order array containing six values per grid node.
- Output data
  - One file in Tecplot binary format containing all data (six variables) for the global grid in column-major order and grid information.

# Optimization

## Original Implementation

- File open, seek, write, close methodology between time steps.
- Headers written element by element. Requires at least 118 writes.

## Optimized Implementation

- File is opened once and remains open during run.
- Headers written by data type or structure. Requires 6 writes.

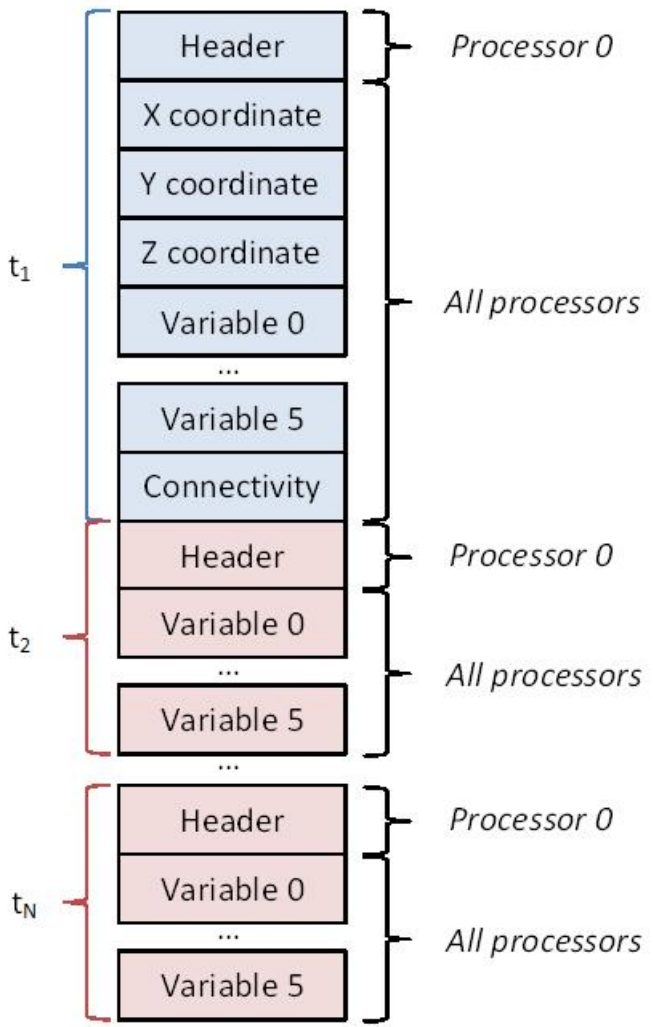



Figure 2: A representation of the Tecplot binary output file format.

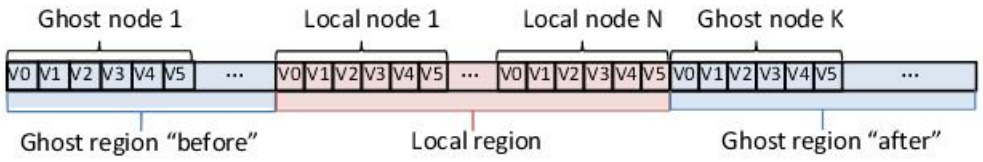
# Optimization

## Original Implementation

- Looping over array indices to determine which to write within data sections.
  - Removal of ghost nodes
  - Separation of variables
- Use of explicit offsets in each data section.

## Optimized Implementation

- Use of derived data types to select portion of array which contains only local region and appropriate variable.
- Use of derived data type to place local data within data section.




- Figure 3: A representation of the local process's data structure. The local and ghost nodes are labeled.

# Results

- Collective MPI-IO calls are utilized along appropriate Collective-buffering and Lustre stripe settings.
  - Stripe count = 160  Stripe Size = 1MB

- Collective MPI-IO calls
  - Account for about a factor of 100 increase in performance.
  - The other optimizations account for about a factor of 2 increase in performance.

**Table 3:** Comparison between Original and Optimized I/O in Application 3

| | | Bandwidth (GB/s) | |
|---|---|---|---|
| Time Step | Data (GB) | Original | Optimized |
| 1 | 1.62 | $3.47 \times 10^{-02}$ | 8.15 |
| 2 | 0.75 | $2.18 \times 10^{-02}$ | 4.22 |
| 3 | 0.75 | $1.91 \times 10^{-02}$ | 5.39 |
| 4 | 0.75 | $1.74 \times 10^{-02}$ | 3.28 |
| 5 | 0.75 | $2.18 \times 10^{-02}$ | 4.54 |
| 6 | 0.75 | $2.05 \times 10^{-02}$ | 3.23 |
| 7 | 0.75 | $2.01 \times 10^{-02}$ | 4.85 |
| 8 | 0.75 | $1.79 \times 10^{-02}$ | 4.56 |
| 9 | 0.75 | $2.59 \times 10^{-02}$ | 4.79 |
| 10 | 0.75 | $2.62 \times 10^{-02}$ | 4.03 |

# Further Information

- Lustre Operations Manual
  - http://dlc.sun.com/pdf/821-0035-11/821-0035-11.pdf
- GPFS:  Concepts, Planning, and Installation Guide
  - http://publib.boulder.ibm.com/epubs/pdf/a7604133.pdf
- HDF5 User Guide
  - http://www.hdfgroup.org/HDF5/doc/PSandPDF/HDF5_UG_r183.pdf
- The NetCDF Tutorial
  - http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-tutorial.pdf

# 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