
# Using MPI-IO environment variables and file striping to allow for large scale CFD simulations

Dr. Vincent Betro

ECSS Symposium: April 15, 2014

# Abstract


- This ECSS project was undertaken by Dr. Vincent Betro with Dr. Diego Donzis of Texas A&M in order to improve the I/O of his CFD code for cyber-enabled investigations of compressible turbulence and mixing and study the effect of thermal non-equilibrium on turbulent processes. In previous work on XSEDE resources, Dr. Donzis developed a new, highly scalable code to perform direct numerical simulations of compressible turbulence and had started obtaining results at resolutions up to 512^3 with a newly developed forcing scheme to maintain a stationary state. Further analysis and new simulations at 1024^3 provided definite answers to pressing important issues about the scaling of different components in which compressible fields can be decomposed, namely solenoidal and dilatational component. The new simulations will be unprecedented in detail and along with the accumulated database, important aspects of small scale intermittency and mixing in compressible turbulence will be, for the first time, investigated. However, this higher resolution also requires more I/O and as a consequence the need for faster parallel I/O.

- In this presentation, Dr. Betro will discuss how MPI_IO environment variables in combination with file striping successfully increased performance. For instance, 32,000 and 64,000 core jobs which could previously not run within a 24 hour walltime can now run successfully with MPI_IO. Also, as one grows the core count, the wall time does not grow as rapidly while still retaining the use of the subarray data types, thus allowing memory use per core to scale better than it could have with a root process having to control all I/O.




NICS

# Outline

- Discuss why large simulations needed
- Discuss importance of using Parallel I/O
- Discuss file striping and MPI-IO
- Discuss environment variables
- Conclusions

# Why do we need to do large CFD simulations?

- Often, the assumptions inherent in RANS (Reynolds Averaged Navier Stokes) fall short when looking at rarefied gases or small scale turbulence (must choose a model, which is inherently presumptive).

- The higher the resolution, the better the accuracy.

- Fluids operate in interestingly predictable ways at the particle level, and in three dimensions, even a small chunk of particles can require billions of “data points”.

# 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

# 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

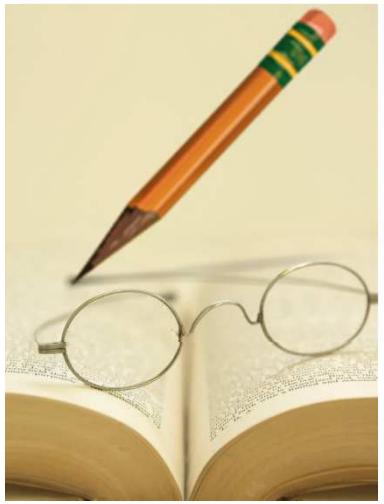
# Scalability Limitation of I/O

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

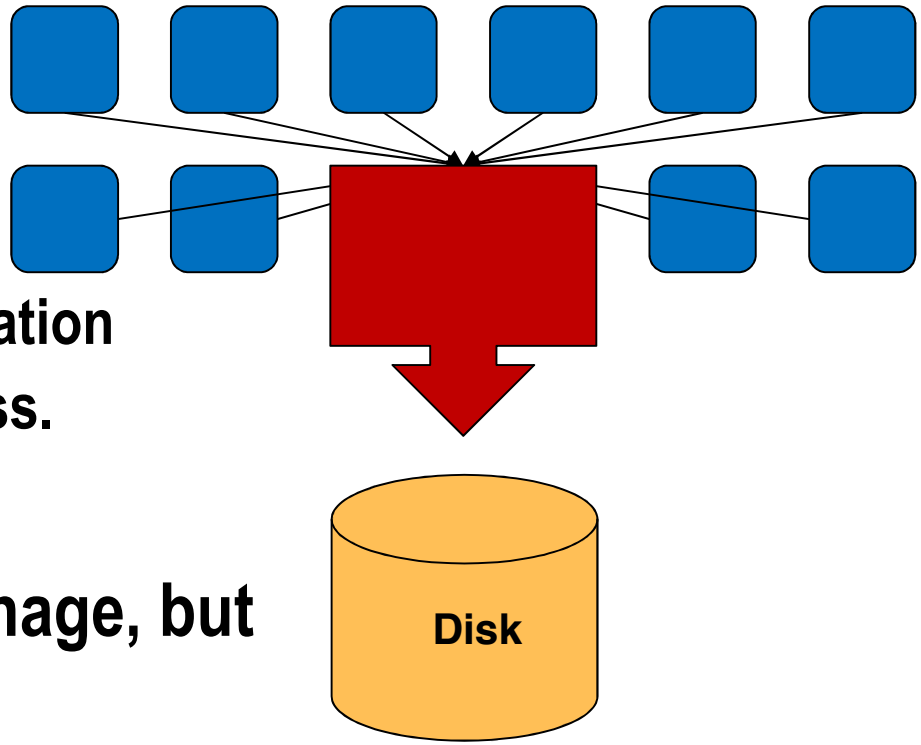
- Once the bandwidth is saturated scaling in I/O stops
  - Adding more compute nodes increases aggregate memory bandwidth and flops/s, but not I/O

# 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).



# Serial I/O: Spokesperson

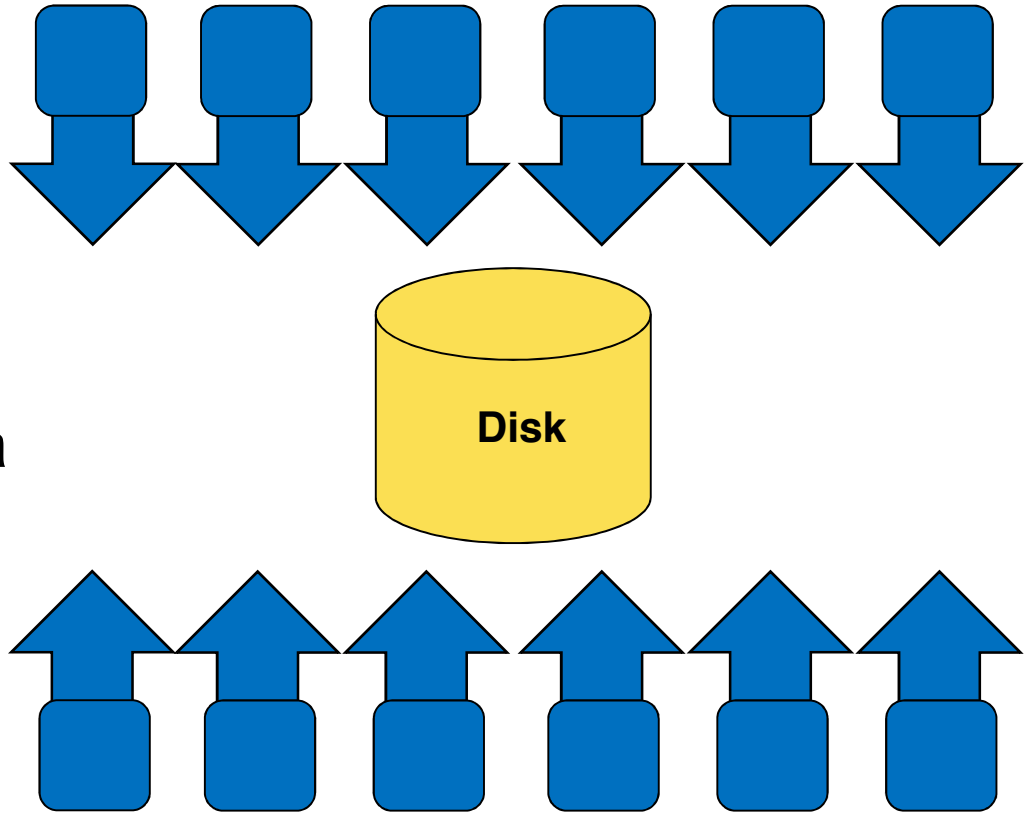



- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.

- 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.

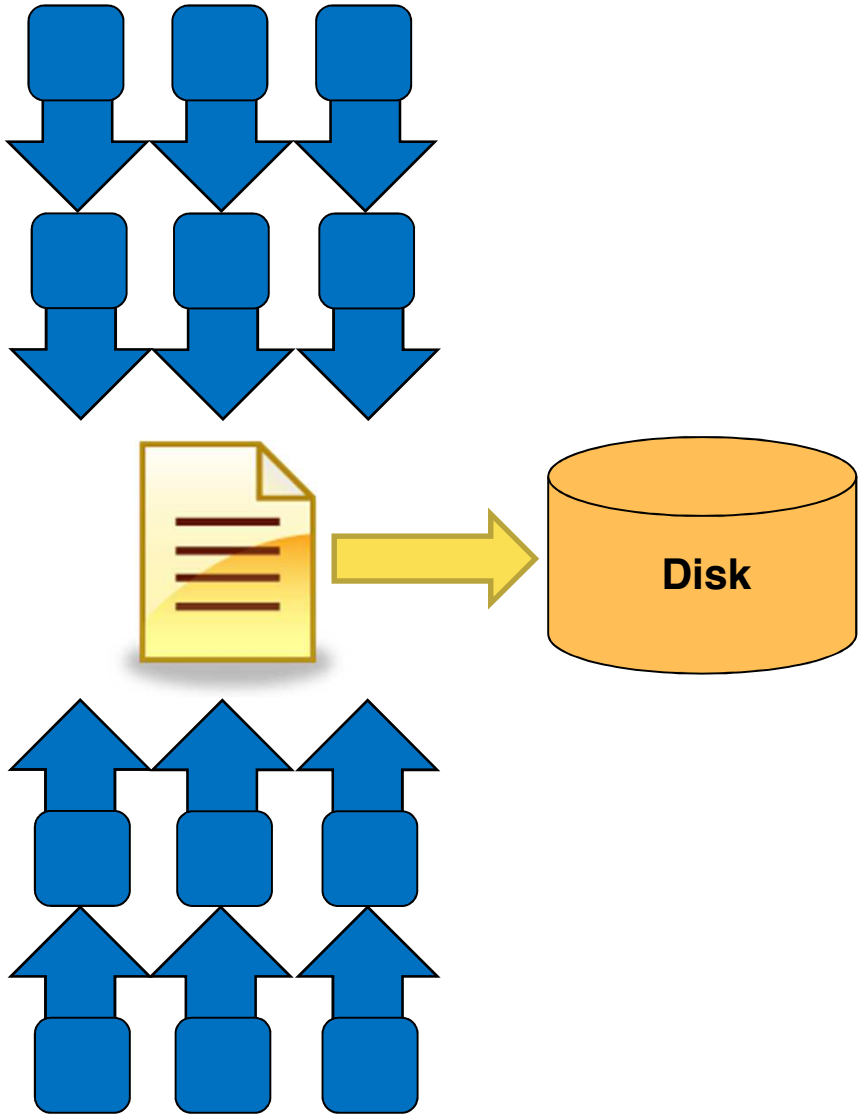





NICS

# 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.





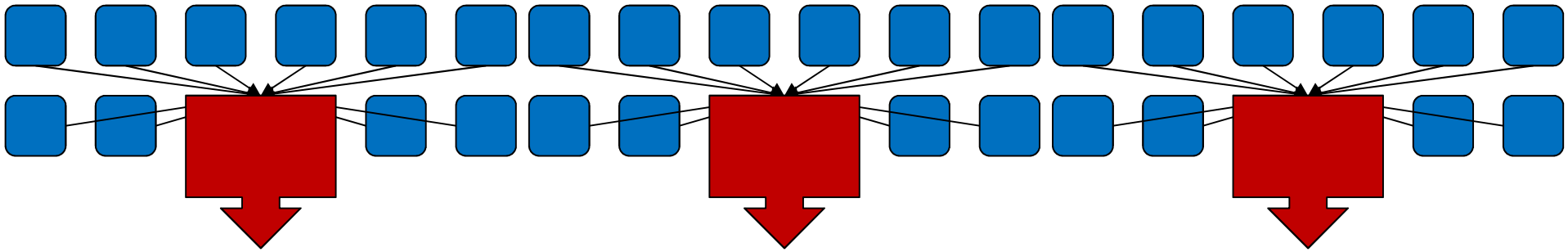

NICS

# 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.



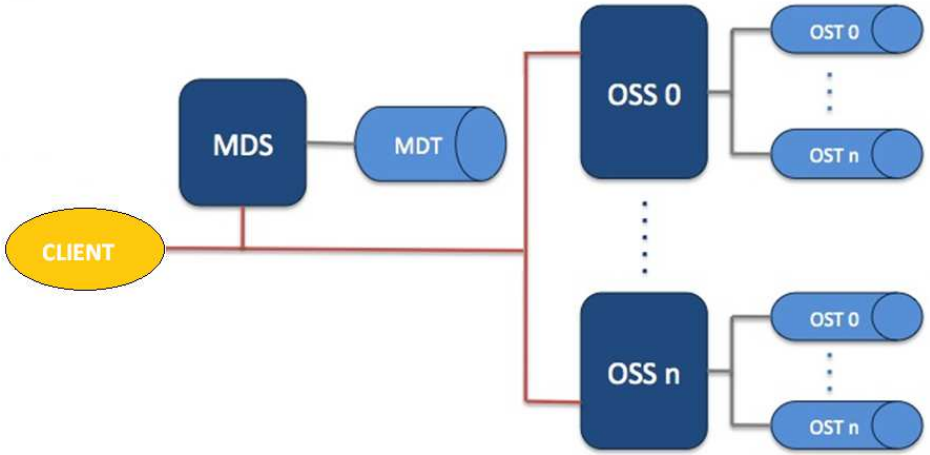
# Performance Mitigation Strategies

- 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.

# File I/O: Lustre File System

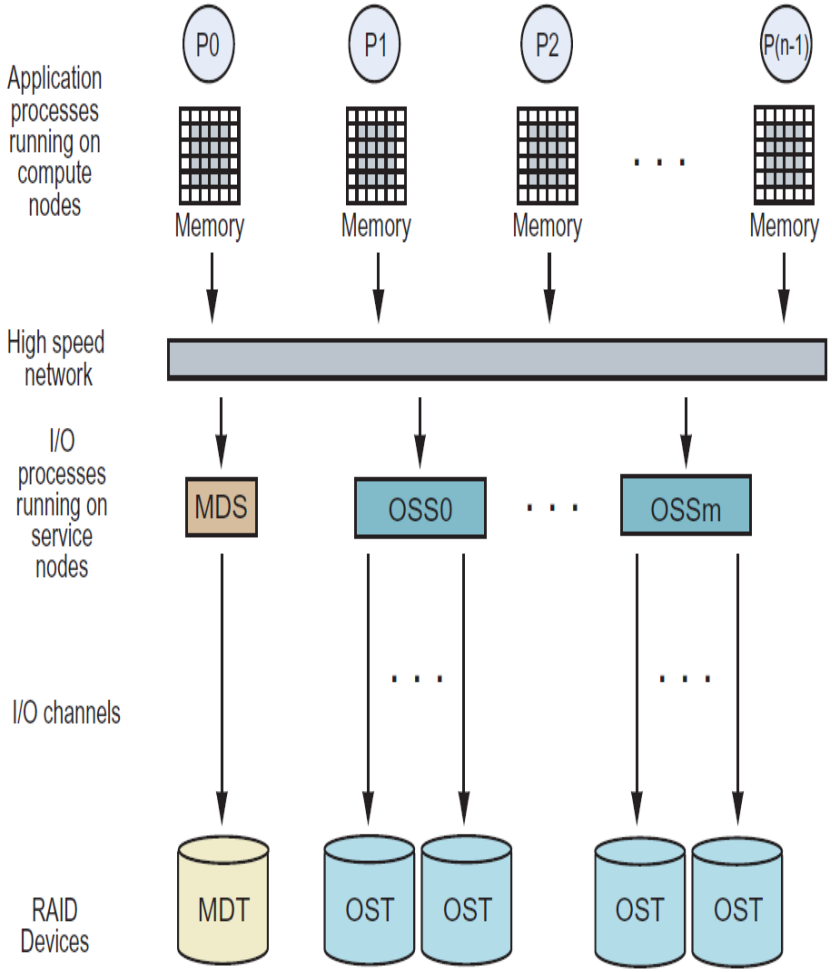
- Metadata Server (MDS) makes metadata stored in the MDT(Metadata Target ) available to Lustre clients.
  - The MDS opens and closes files and stores directory and file Metadata such as file ownership, timestamps, and access permissions on the MDT.
  - Each MDS manages the names and directories in the 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).



# 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

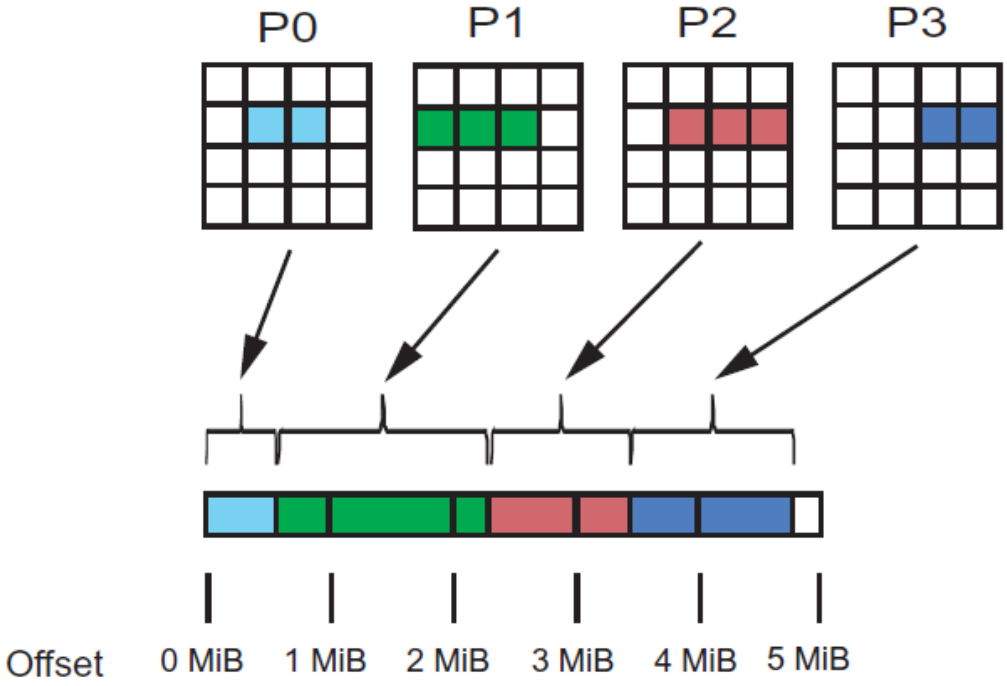
- A single file may be stripped across one or more OSTs (chunks of the file will exist on more than one OST).

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

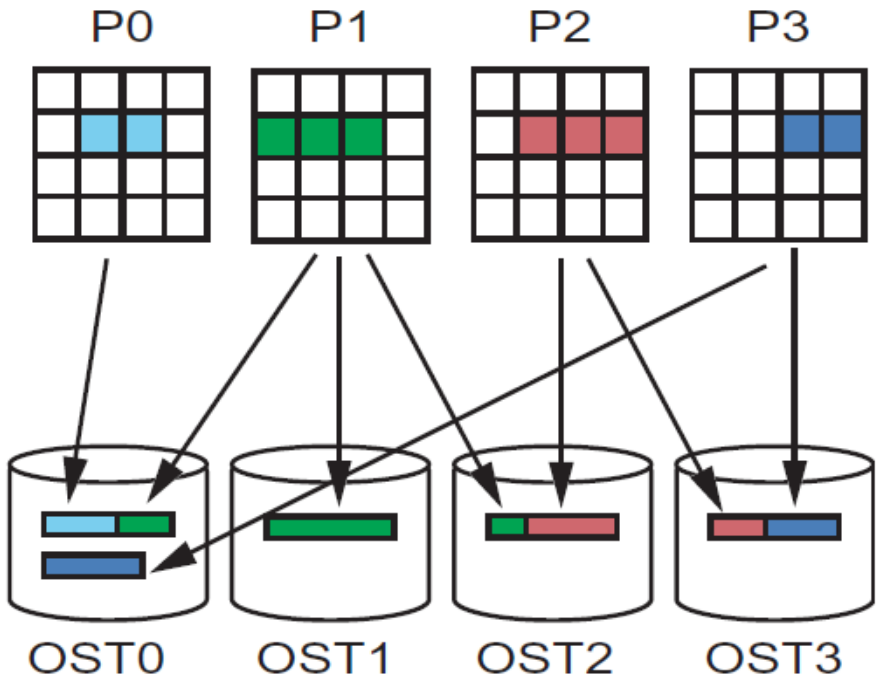
- Disadvantage:
  - increased overhead due to network operations and server contention

→ 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).**

# 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).

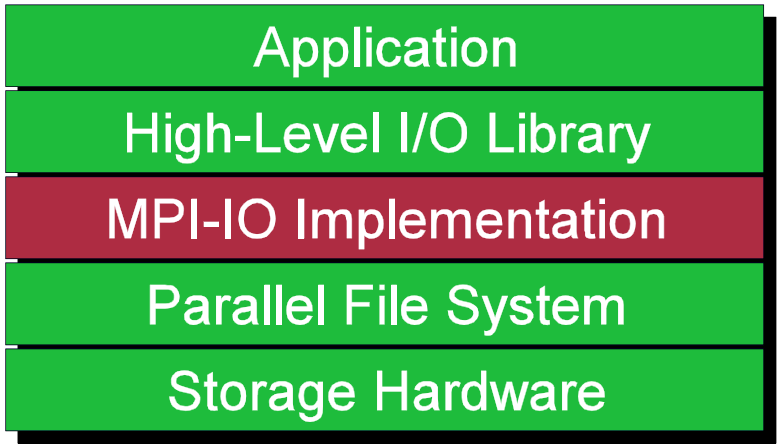
# Scalability

- 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.

# 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)

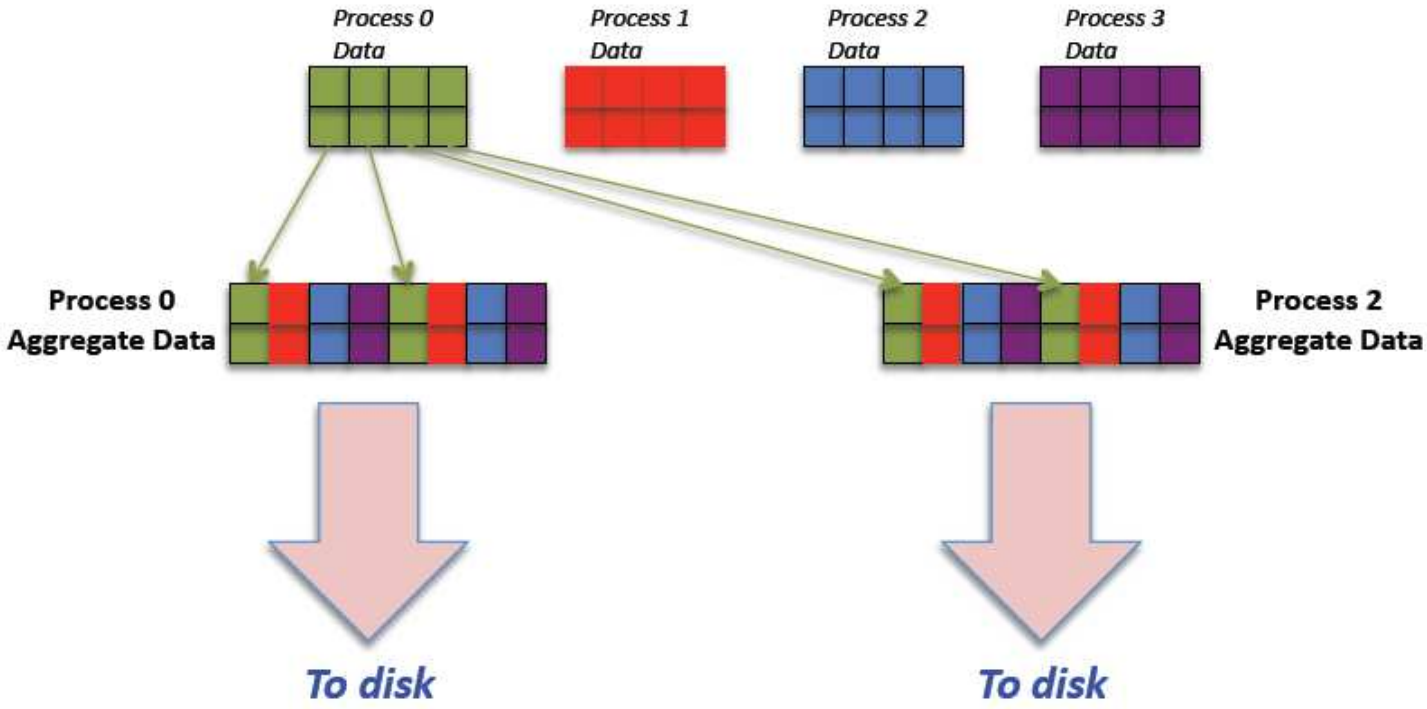
```
/* Open the file */
MPI_File_open(MPI_COMM_WORLD,'file', MPI_MODE_RDONLY, MPI_INFO_NULL,&fh);

/* 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);
```

# 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.

# 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.

# Environment variables for Kraken

- Under the original MPI_IO implementation, a job with 8K cores runs in 159,755,913 seconds; a job with 16K cores runs in 466,281,565 seconds (which is a 3x SLOWDOWN); and a job with 32K cores runs out of memory. This is counter to the purpose of MPI_IO having been added instead of having the root process collect all the data (which is actually much faster than the original MPI_IO implementation). In fact, by implementing the MPI_IO collective communication, all processes reading/writing simultaneously was placing an insurmountable amount of work on the OSTs for very large processor counts before collective buffering was added.

# Environment variables for Kraken

- In order to combat this slowdown, an attempt was made to see if simple file striping would correct the issue, but after trying stripe counts of 2, 4, 8, 16, 32, and all OSTs with no noticeable gain in performance, both this and a growth in the MPICH_UNEX_EVENT_BUFFER were scrapped as simple solutions. After testing the code with several sequential read and write calls, which slowed things down even more, we attempted to implement a collective buffering type operation manually (using gather and scatter), thinking that it would be more optimized than simply using the environment variables. Instead, it lead to code runtime errors where the rank order things were being scattered and gathered by in the Cartesian row or column communicator was not the same as that needed by the data subarray type created for MPI_IO from the original communicator.

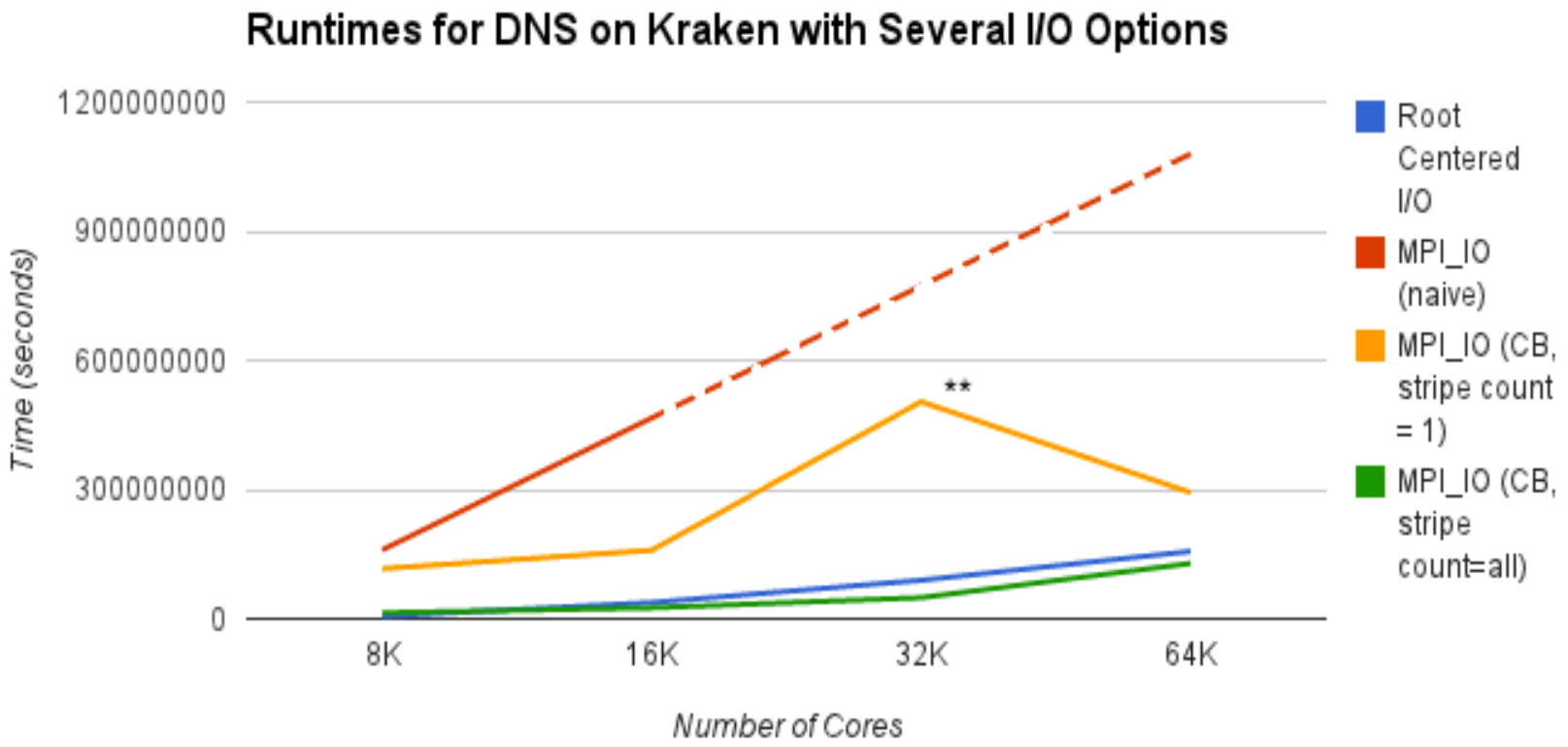
# Environment variables for Kraken

- So, by using the following MPI_IO environment variables in combination with file striping (a stripe count of 1 significantly slowed the MPI_IO, but setting stripe to all OSTs (-c -1) ran faster than the multi-file version), we were able to successfully increase performance such that A) 32,000 and 64,000 core jobs can now run successfully with MPI_IO (and faster than with the original multi-file implementation) and B) as one grows the core count, the wall time does not grow as rapidly. Also, we were able to retain the use of the subarray data types, thus allowing memory use per core to scale better than with a root process having to control all I/O.

# Environment variables for Kraken

| MPICH_MPIIO_HINTS (to all files read and written) | |
|---|---|
| Romio_cb_write | Enable |
| Romio_ds_write | Disable (deprecated call) |
| Romio_cb_read | Enable |
| Romio_ds_read | Disable (deprecated call) |
| MPICH_MPIIO_HINTS_DISPLAY | 1 (DEBUG ONLY) |
| MPICH_MPIIO_XSTATS | 1 (DEBUG ONLY) |
| MPICH_MPIIO_CB_ALIGN | 1 (aligns with lustre striping/boundaries) |
| MPICH_PTL_MATCH_OFF | 1 (disables registration of recv requests w/portals) (to overcome PtlMEMDPost() failed: PTL_NO_SPACE) |
| Lfs setstripe outdata –c -1 | Set from head directory |

# Outcomes on Kraken




*Values for naïve MPI_IO are extrapolated due to their inability to run.

** This value from MPI_IO with stripe count of 1 is likely due to a heavy load at that moment on that OST, not the actual core count.

# Contact Information


- Dr. Vincent Charles Betro
- University of Tennessee National Institute for Computational Sciences

  Oak Ridge National Laboratory

  Bldg 5100, Room 247

  P.O. Box 2008 MS 6173

  Oak Ridge, TN 37831
- Phone: 865-576-8905
- email: vbetro@tennessee.edu



A SPECIAL THANKS TO BILEL HADRI FOR THE WONDERFUL PICTURES AND SLIDES ABOUT PARALLEL I/O SO I DIDN'T HAVE TO REINVENT THE WHEEL!

# Questions?