Data Intensive Computing and Parallel I/O

Ritu Arora

(With Contributions from Dan Stanzione, Doug James, John Cazes, Robert McLay, Jessica Trelogan)

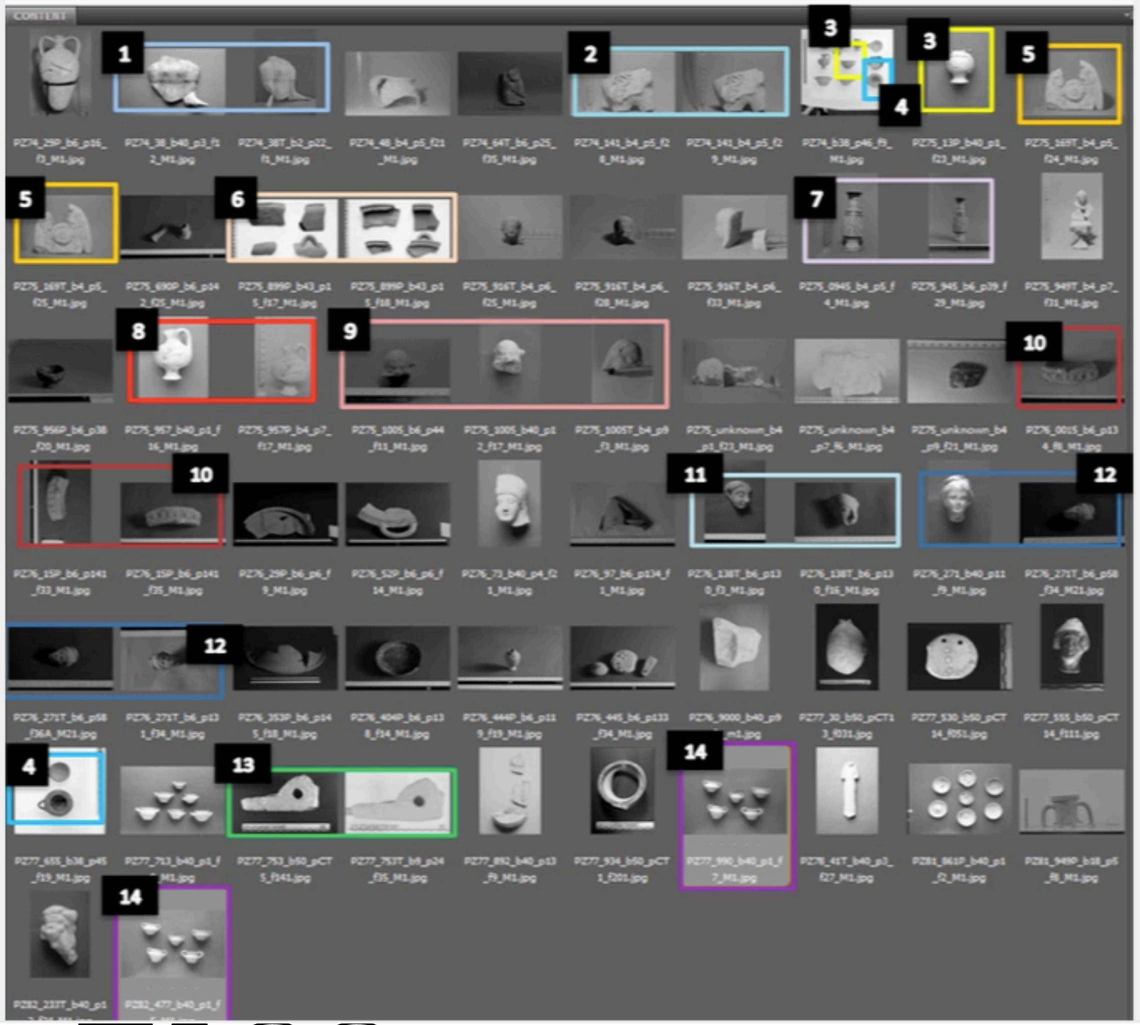
Email: **rauta@tacc.utexas.edu**

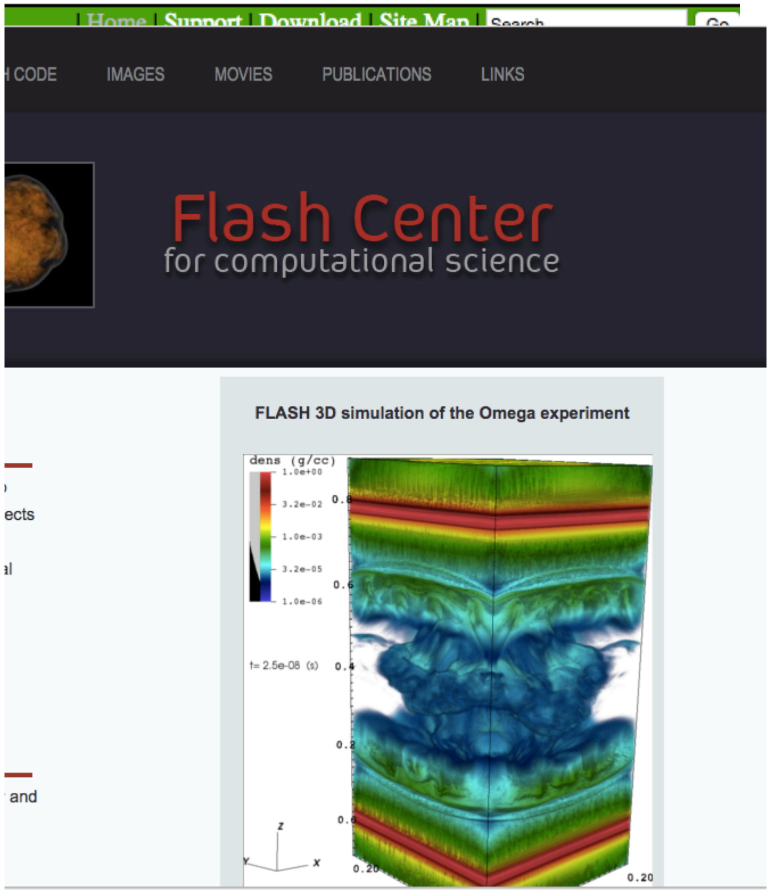
Overview

- Introduction to Data Intensive Computing and application I/O
- I/O During Pre-Processing
  - Choose the right data transfer protocol
  - Selective I/O
- I/O During Processing
  - Understand your parallel file system
  - Use parallel I/O
- I/O During Post-Processing
  - Move your data to secondary or tertiary storage media
- An Example of a System Specifically Designed for Data Intensive Computing
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

Data Intensive Computing

mpiBLAST: Open-Source Parallel BLAST




Home | Support | Download | Site Map
CODE
IMAGES
MOVIES
PUBLICATIONS
LINKS
Flash Center
for computational science
FLASH 3D simulation of the Omega experiment
dens (g/cc)

THE UNIVERSITY OF TEXAS AT AUSTIN

Application I/O

- Software applications often
  - Read initial conditions or datasets for processing
  - Write numerical data from simulations
    - Saving application-level checkpoints

- The total execution time of an application can be broken down into the computation time, communication time, and the I/O time

- Optimizing the time spent in computation, communication and I/O can lead to overall improvement in the application performance

- However, doing efficient I/O without stressing out the HPC system is challenging and often an afterthought

Prior to Running Data Intensive Applications on HPC Systems at Open-Science Datacenters

- Prepare a data management plan
  - Determine the type of data, amount of data, and the rate at which the data will be produced or consumed
  - Determine the data retention value
  - Identify the required hardware and software for storing and accessing the data
  - Be aware of any compliance needs or policies for data usage
- Learn about the usage policies associated with the systems that you would like to use
  - Know your filesystem
  - Know about do's and don'ts on the resources of interest
- A sample checklist for data management plan:

  http://www.dcc.ac.uk/resources/data-management-plans/checklist

Overview

- Introduction to Data Intensive Computing and application I/O
- **I/O During Pre-Processing Stage**
  - **Choose the right data transfer protocol**
  - Selective I/O
- I/O During Processing Stage
  - Understand your parallel file system
  - Use parallel I/O
- I/O During Post-Processing Stage
  - Move your data to secondary or tertiary storage media
- An Example of a System for Data Intensive Computing
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

Protocols for Data Transfer

- Different protocols exist for data transfer to (and between) remote sites, *e.g.*,
  1. Linux command-line utilities **`scp`** & **`rsync`**
  2. Globus' **`globus-url-copy`** command-line utility
  3. Globus Connect

- Check the user-guide of the resource that you are wanting to use to see the list of supported protocols

Data Transfer Using **`scp`** or WinSCP

- If your local computer is a Mac or a Linux laptop, you can use the `scp` commands to transfer data to and from a remote resource like Stampede

```
scp filename username@hostname:/path/to/directory
```

- If you are using a Windows computer, you can download and use the WinSCP application (GUI-based), or download and use Cygwin (command-line based, can run the aforementioned commands)
  – For small amounts of data, you may also use the "File Transfer Window" available in the SSH client – drag an drop the files across the local laptop and a remote resource

More Information on Using WinSCP

- For learning the usage of WinSCP the following slides and video might be useful for the Windows users

- Slides:
  https://drive.google.com/open?id=0B8zOSeBE0p0rUDZvVVR4aHl5b0k

- Video:
  https://www.youtube.com/watch?v=Nn7Ofb0lYwM

Data Transfer Using **rsync**

- The `rsync` command is another way to transfer data and to keep the data at the source and destination in sync

```
rsync path-to-source-file path-to-destination-file
```

- If transferring the data for the first time to a remote resource, `rsync` and `scp` might show similar performance except when the connection drops
  - If a connection drops, upon restart of the data transfer, `rsync` will automatically transfer only the remaining files to the destination, it will skip the already transferred files
- `rsync` transfers only the actual changed parts of a file (instead of transferring an entire file)
  - this selective method of data transfer can be much more efficient than `scp` because it reduces the amount of data sent over the network

Using Globus Connect

- Globus Connect provides fast, secure transport via an easy-to-use web interface using pre-defined and user-created "endpoints"

- Globus Connect makes it possible to create a transfer endpoint on any machine (including campus servers and home laptops) with few clicks

- For more information on Globus Connect:

  https://www.globus.org/globus-connect

  http://www.cac.cornell.edu/vw/DataTransfer/globus.aspx

Data Transfer Issues – Real World Scenario

- During one project, transferring **4.3 TB** of data from the Stampede Supercomputer in Austin to the Gordon Supercomputer in San Diego, took **approx. 210 hours**
- The transfer was **restarted about 14 times** during June 3 to June 18, 2014 - about 15 days
- If the data transfer would have completed without any interruptions, it would have completed in about 9 days at the given speed
- Multiple reasons for interruption - sometimes maintenance on Stampede or Gordon, some other file-system issue, network traffic/available bandwidth - all are factors affecting the data transfer rate

Overview

- Introduction to Data Intensive Computing and application I/O
- **I/O During Pre-Processing Stage**
  - Choose the right data transfer protocol
  - **Selective I/O**
- I/O During Processing Stage
  - Understand your parallel file system
  - Use parallel I/O
- I/O During Post-Processing Stage
  - Move your data to secondary or tertiary storage media
- An Example of a System for Data Intensive Computing
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

Culling the Data Collection

Eliminate undesirable data

Improve quality

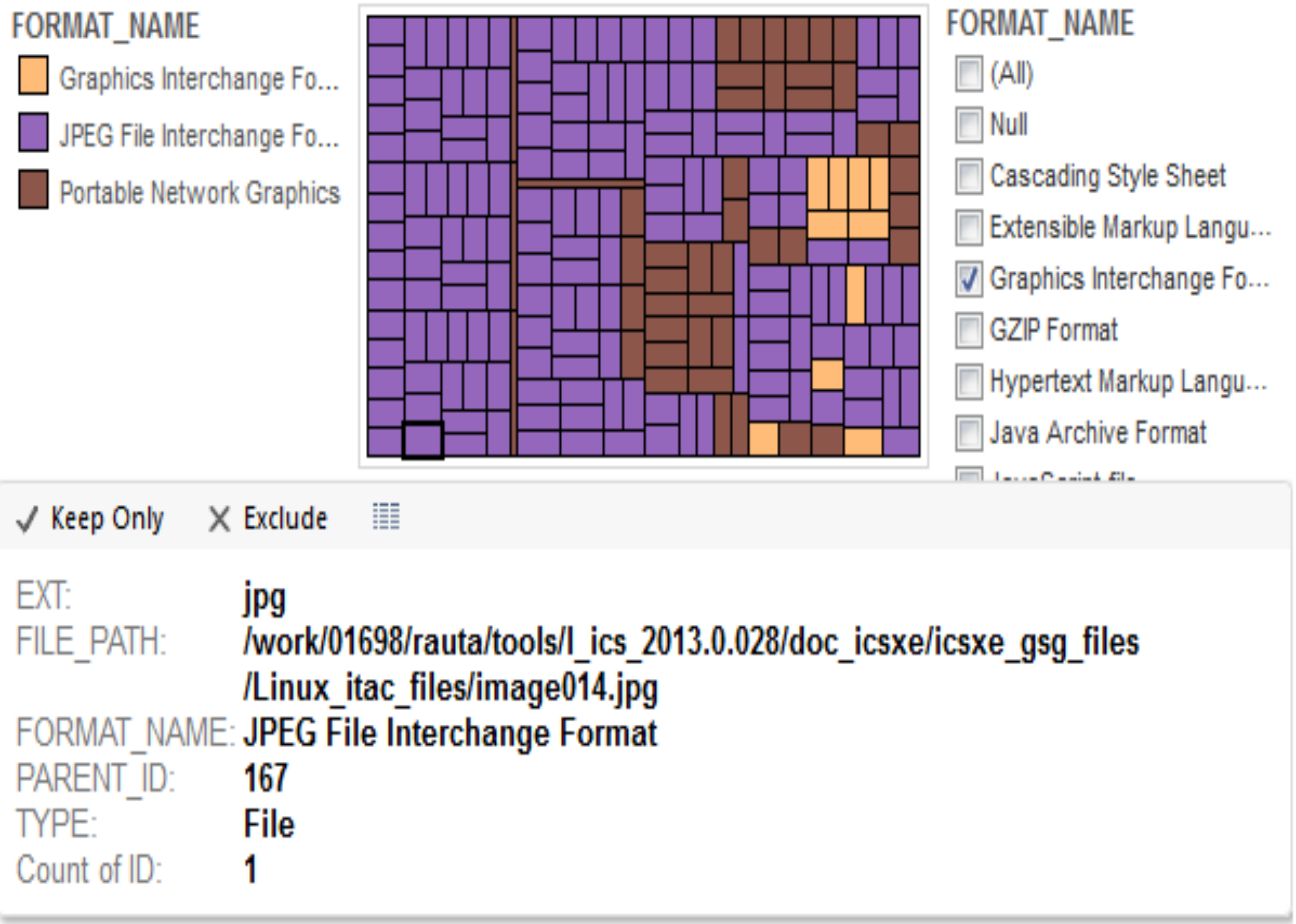
Reduce processing needs

- Filtering
  - By name, size, date
  - By type
  - By author, tags, metadata
- De-duplication
  - Exact binary match by checksum
- <u>Required</u>
  - Culling algorithm
  - Scripts for automation
  - Human intelligence

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

Interactive Visualization for Data Culling and Quality Control

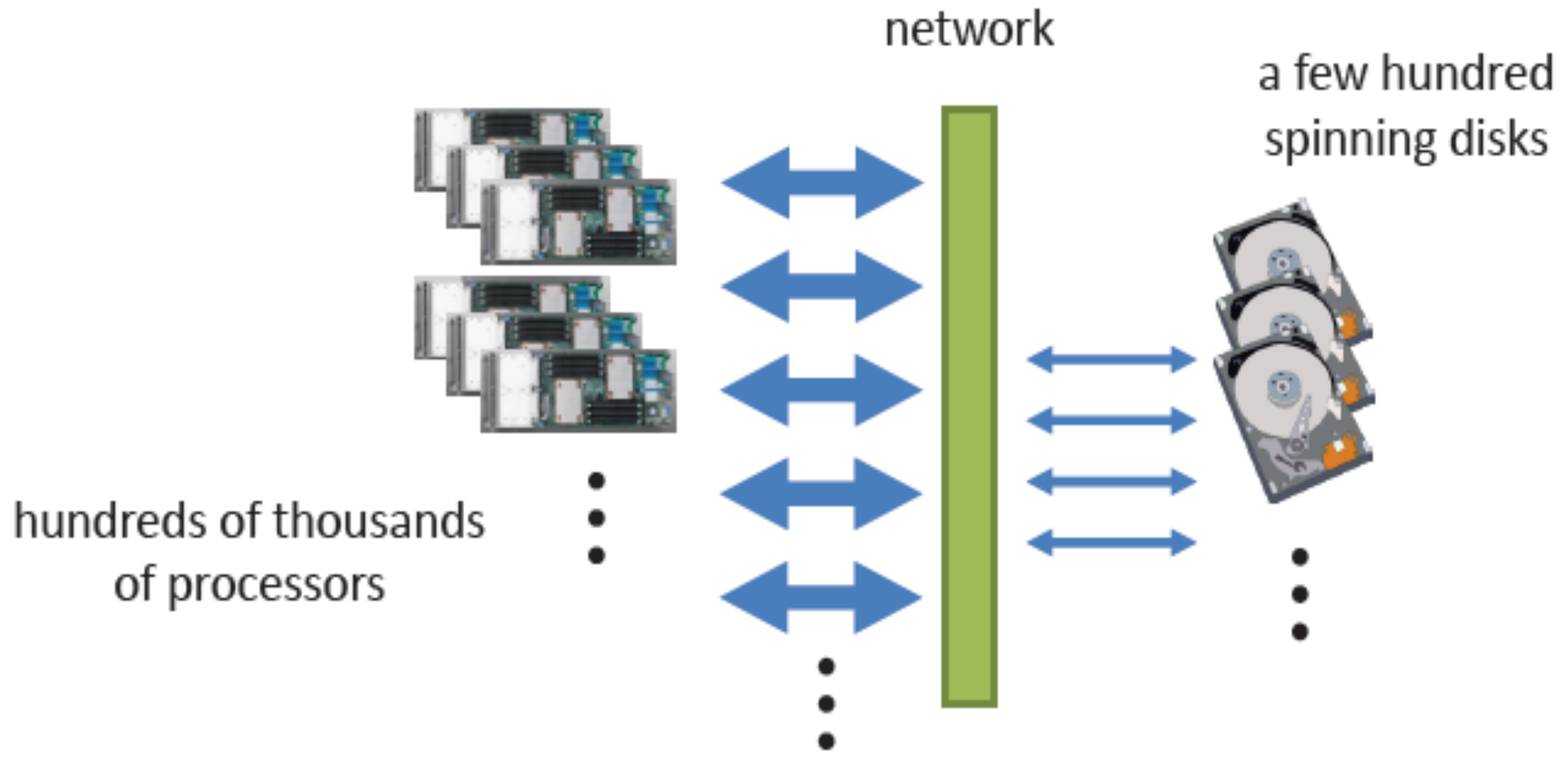

FORMAT_NAME
🟧 Graphics Interchange Fo...
🟪 JPEG File Interchange Fo...
🟫 Portable Network Graphics
FORMAT_NAME
☐ (All)
☐ Null
☐ Cascading Style Sheet
☐ Extensible Markup Langu...
☑ Graphics Interchange Fo...
☐ GZIP Format
☐ Hypertext Markup Langu...
☐ Java Archive Format
✓ Keep Only
X Exclude

EXT:	jpg
FILE_PATH:	/work/01698/rauta/tools/l_ics_2013.0.028/doc_icsxe/icsxe_gsg_files /Linux_itac_files/image014.jpg
FORMAT_NAME:	JPEG File Interchange Format
PARENT_ID:	167
TYPE:	File
Count of ID:	1

Overview

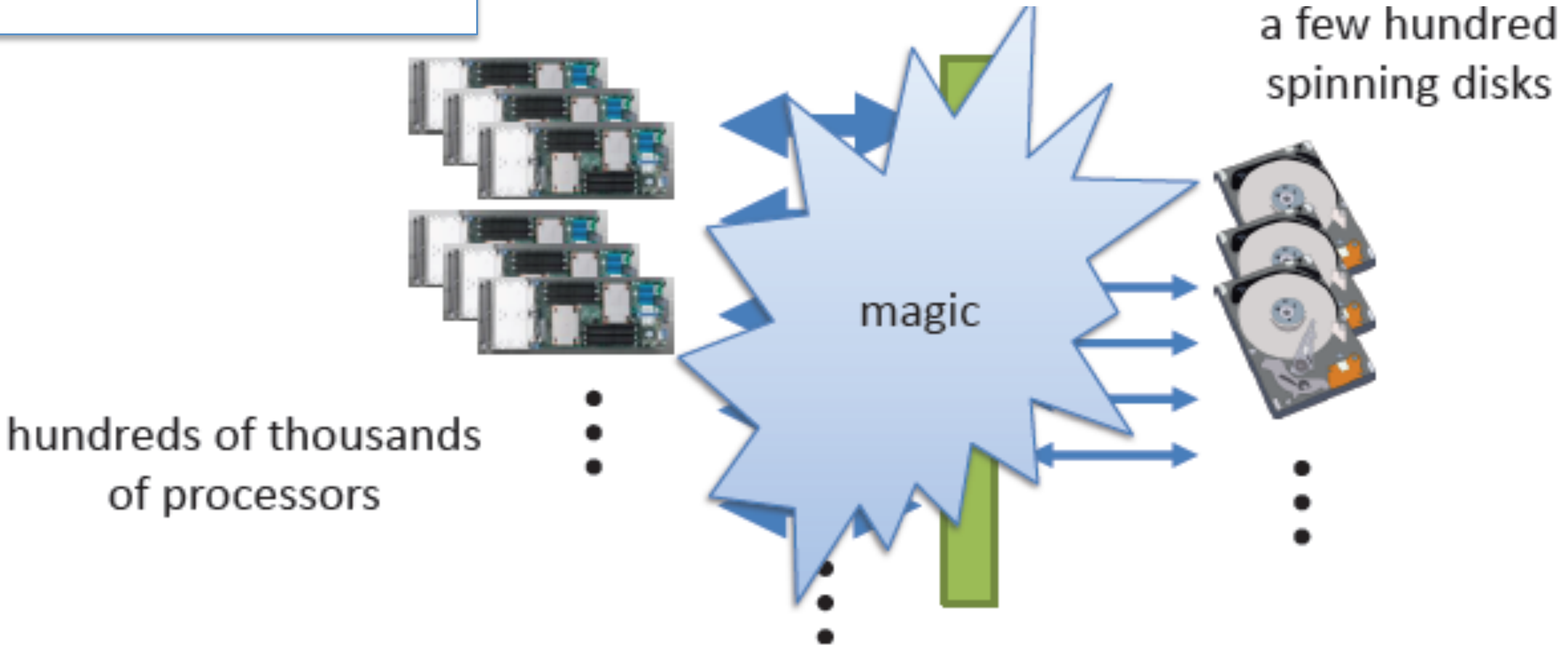
- Introduction to Data Intensive Computing and application I/O
- I/O During Pre-Processing Stage
  - Choose the right data transfer protocol
  - Selective I/O
- **I/O During Processing Stage**
  - **Understand your parallel file system**
  - Use parallel I/O
- I/O During Post-Processing Stage
  - Move your data to secondary or tertiary storage media
- An Example of a System for Data Intensive Computing
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

Modern HPC Cluster

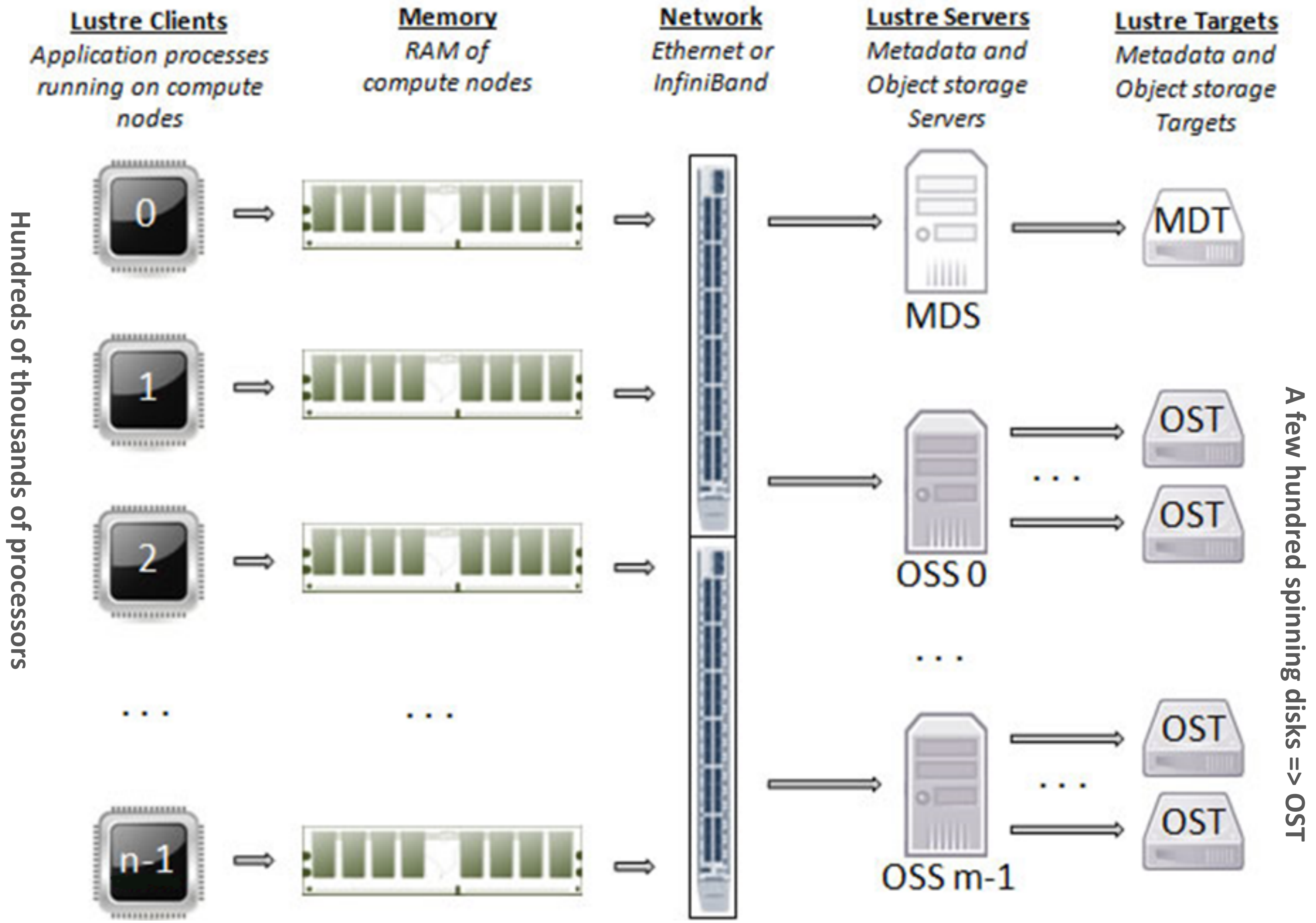

network
a few hundred spinning disks
hundreds of thousands of processors

Modern HPC Cluster

…we need some magic to make the collection of spinning disks act like a single disk for the user…


a few hundred spinning disks
magic
hundreds of thousands of processors

…Parallel Filesystem (*e.g.*, Lustre) Provides the Magic


Lustre Clients
Application processes running on compute nodes
Memory
RAM of compute nodes
Network
Ethernet or InfiniBand
Lustre Servers
Metadata and Object storage Servers
Lustre Targets
Metadata and Object storage Targets
Hundreds of thousands of processors
0
1
2
. . .
n-1
MDS
MDT
OSS 0
OSS m-1
OST
A few hundred spinning disks => OST

Source: Reference 2, 4

Lustre Filesystem at TACC

- Each Lustre filesystem has a different number of OSTs
- The greater the number of OSTs the better the I/O capability
- To check the number of OSTs available on the filesystems, you may use the command:

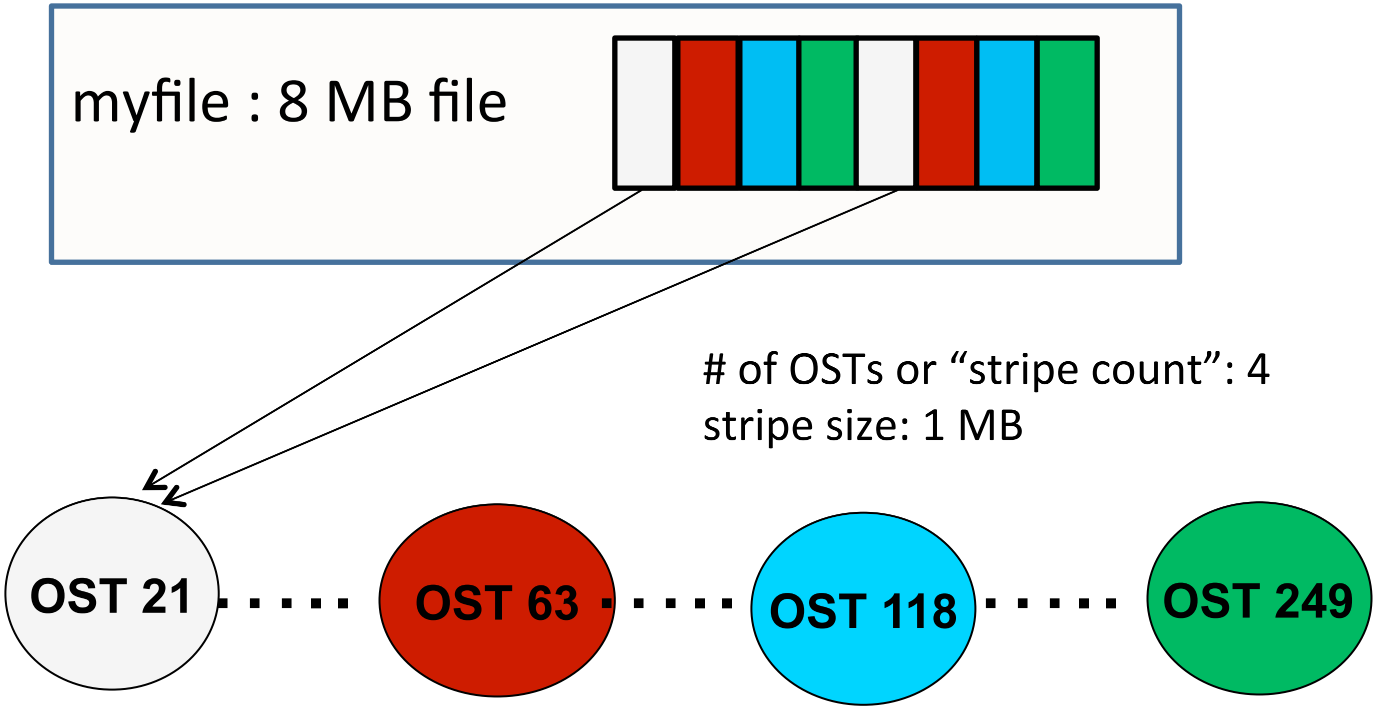
```
$ lfs osts
```

	$HOME	**$WORK**	**$SCRATCH**
Stampede 1.0	24	672	348

Lustre File System - Striping

- Lustre supports the striping of files across several I/O servers (similar to RAID 0)

- Each stripe is a fixed size block



Lustre File System – Striping on TACC Resources

- Administrators set a default stripe count and stripe size that applies to all newly created files
  - Stampede 1.0: `$SCRATCH`: 2 stripes/1MB
    `$WORK`: 1 stripe /1MB

- However, users can reset the default stripe count or stripe size using the Lustre commands

Lustre Commands

- Get stripe count

```
 % lfs getstripe ./testfile
./testfile
lmm_stripe_count:   2
lmm_stripe_size:    1048576
lmm_stripe_offset:  50
        obdidx           objid          objid            group
            50         8916056       0x880c58                0
            38         8952827       0x889bfb                0
```

- Set stripe count

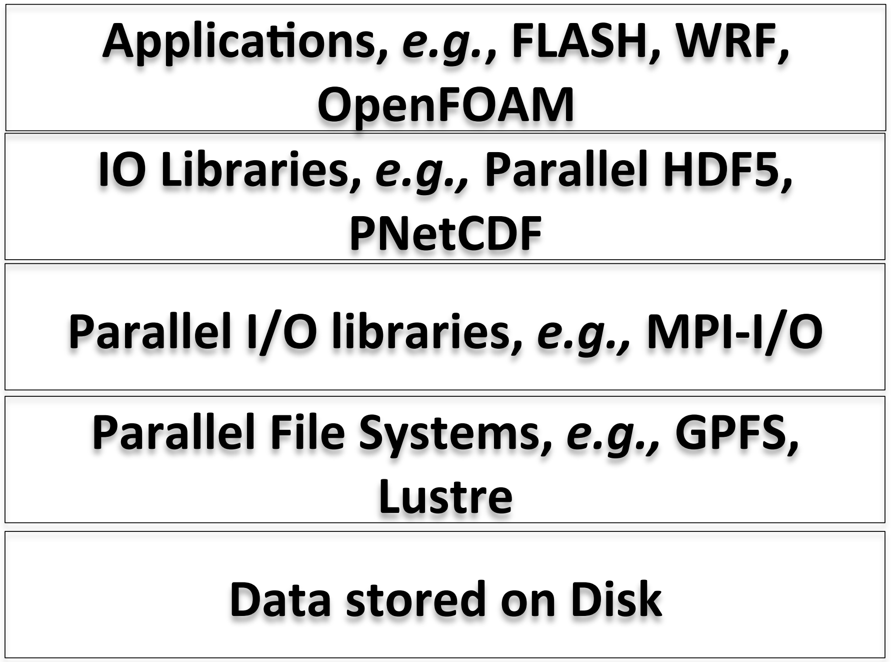
```
% lfs setstripe -c 4 -s 4M testfile2
% lfs getstripe ./testfile2
./testfile2
lmm_stripe_count:   4
lmm_stripe_size:    4194304
lmm_stripe_offset:  21
        obdidx           objid          objid            group
            21         8891547       0x87ac9b                0
            13         8946053       0x888185                0
            57         8906813       0x87e83d                0
            44         8945736       0x888048                0
```

Real-World Scenario
FLASH code: impact of file striping on I/O

LFS Stripe Count #	Time taken for reading a checkpoint (in seconds)	Time Taken for Writing a Checkpoint (in seconds)
2	515.528	494.212
30	61.182	175.892
40	53.445	108.782
60	46.913	182.65
80	40.57	183.107

Need for High-Level Support for Parallel I/O

- Parallel I/O can be hard to coordinate and optimize if working directly at the level of Lustre API
- Therefore, specialists implement a number of intermediate layers for coordination of data access and mapping from application layer to I/O layer
- Hence, application developers only have to deal with a high-level interface built on top of a software stack, that in turn sits on top of the underlying hardware
  - *e.g.,* MPI-I/O, parallel HDF5, T3PIO


Applications, *e.g.,* FLASH, WRF, OpenFOAM
IO Libraries, *e.g.,* Parallel HDF5, PNetCDF
Parallel I/O libraries, *e.g.,* MPI-I/O
Parallel File Systems, *e.g.,* GPFS, Lustre
Data stored on Disk

Implementation Layers

See Reference # 4

You Can Stress Out Lustre Easily if You…

- Open and close the same file every few milliseconds
  - Stresses the MDS
- Too often, too many
  - Stresses the MDS and OSTs
- Write large files to `$HOME` or `$WORK`
  - `$SCRATCH` should be used instead of `$HOME` or `$WORK`
- `ls` in a crowded directory
  - `ls` is aliased to “`ls --color=tty`"
  - Every directory item incurs the overhead of an extra “stat" call to the MDS
  - Use /bin/ls in a crowded directory
- Create thousands of files in the same directory
  - A directory too is a file managed by the MDS

What happens when Lustre gets stressed out?

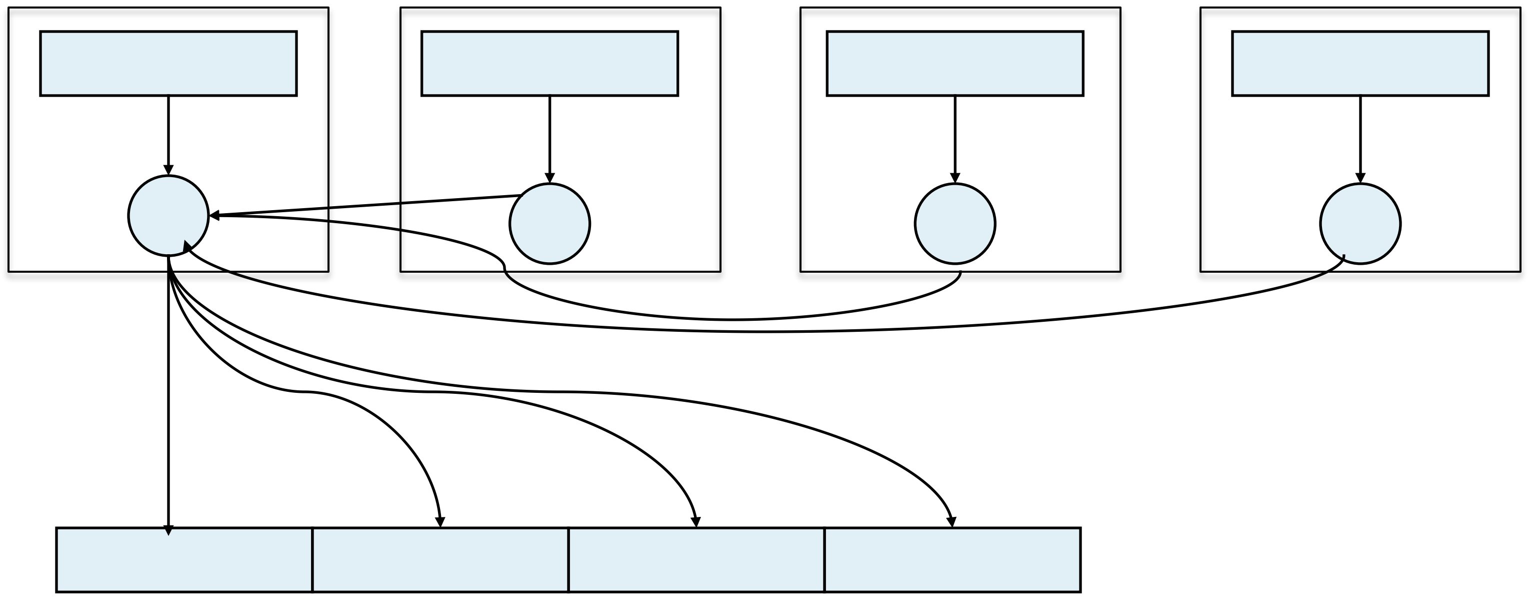
Overview

- Introduction to Data Intensive Computing and application I/O
- I/O During Pre-Processing Stage
  - Choose the right data transfer protocol
  - Selective I/O
- **I/O During Processing Stage**
  - Understand your parallel file system
  - **Use parallel I/O**
- I/O During Post-Processing Stage
  - Move your data to secondary or tertiary storage media
- An Example of a System for Data Intensive Computing
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

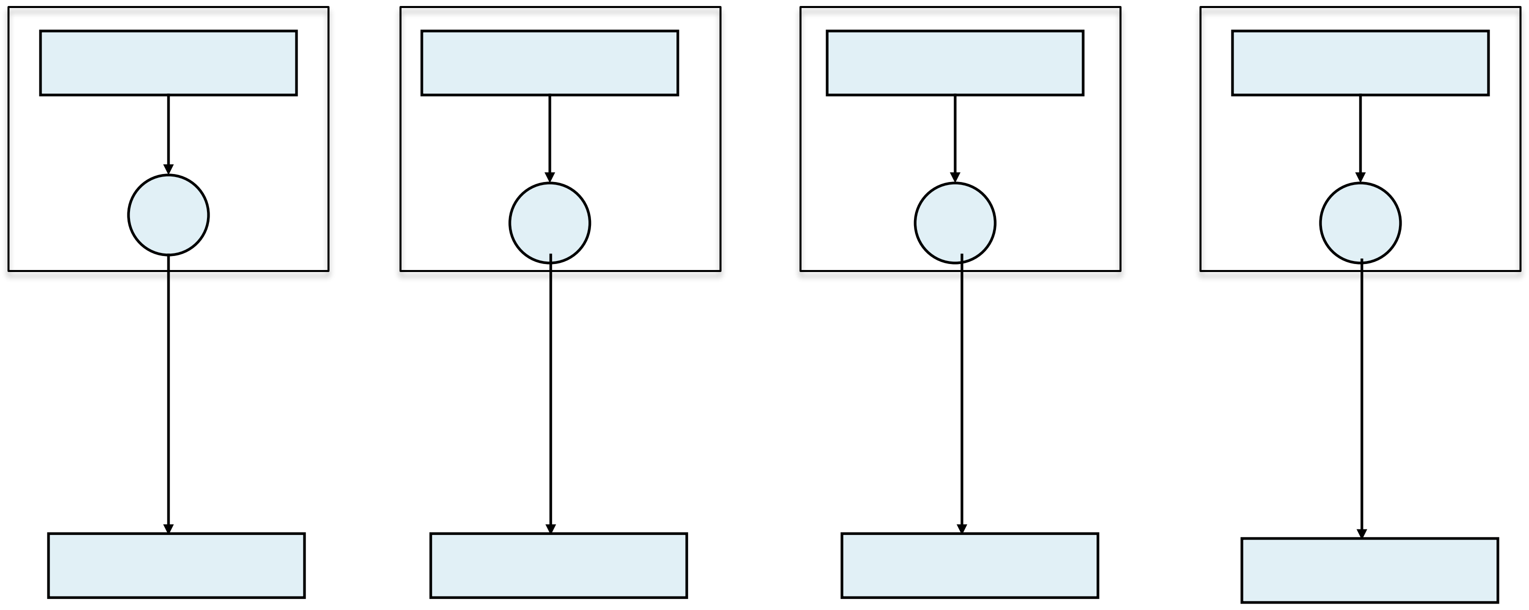
Typical Pattern: Parallel Programs Doing Sequential I/O

- All processes send data to master process, and then the process designated as master writes the collected data to the file

- This sequential nature of I/O can limit performance and scalability of many applications

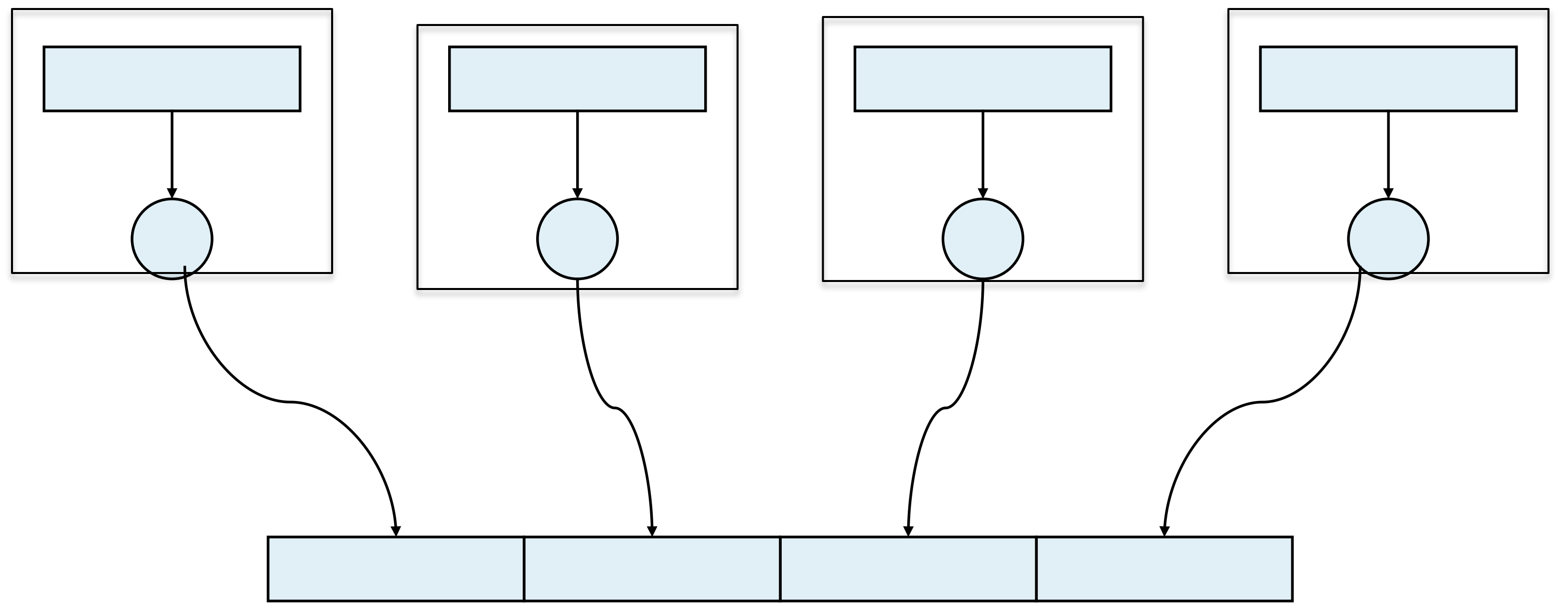


Another Pattern: Each Process Writing to a Separate File



Desired Pattern: Parallel Programs Doing Parallel I/O

- Multiple processes participating in reading data from or writing data to a common file in parallel
- This strategy improves performance and provides a single file for storage and transfer purposes

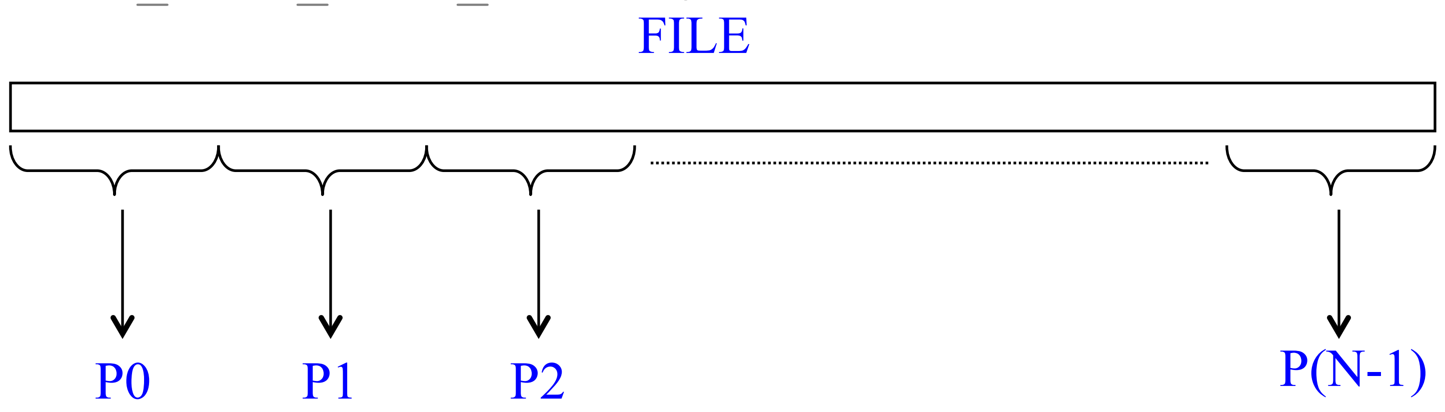


MPI for Parallel I/O

- A parallel I/O system for distributed memory architectures will need a mechanism to specify collective operations and specify noncontiguous data layout in memory and file
- Reading and writing in parallel is like receiving and sending messages
- Hence, an MPI-like machinery is a good setting for Parallel I/O (think MPI communicators and MPI datatypes)
- MPI-I/O featured in MPI-2 which was released in 1997, and it interoperates with the filesystem to enhance I/O performance for distributed-memory applications

Using MPI-I/O

- Given N number of processes, each process participates in reading or writing a portion of a common file
- There are three ways of positioning where the read or write takes place for each process:
  - Use individual file pointers (*e.g.,* `MPI_File_seek`/`MPI_File_read`)
  - Calculate byte offsets explicitly (*e.g.,* `MPI_File_read_at`)
    - Explicit offset operations perform data access at the file position given directly as an argument — no file pointer is used nor updated
  - Access a shared file pointer (*e.g.,* `MPI_File_seek_shared`, `MPI_File_read_shared`)



Source: Reference 3

MPI-I/O API Opening and Closing a File

- Calls to the MPI functions for reading or writing must be preceded by a call to `MPI_File_open`
  - `int` **`MPI_File_open`**`(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)`
- The parameters below are used to indicate how the file is to be opened

`MPI_File_open` mode	**Description**
`MPI_MODE_RDONLY`	read only
`MPI_MODE_WRONLY`	write only
`MPI_MODE_RDWR`	read and write
`MPI_MODE_CREATE`	create file if it doesn't exist

- To combine multiple flags, use bitwise-or " | " in C, or addition "+" in Fortran
- Close the file using: **`MPI_File_close`**`(MPI_File fh)`

MPI-I/O API for Reading Files

After opening the file, read data from files by either using `MPI_File_seek` & `MPI_File_read` Or `MPI_File_read_at`

```
int MPI_File_seek( MPI_File fh, MPI_Offset offset,
int whence )
int MPI_File_read(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

`whence` in **`MPI_File_seek`** updates the individual file pointer according to

`MPI_SEEK_SET`: the pointer is set to offset

`MPI_SEEK_CUR`: the pointer is set to the current pointer position plus offset

`MPI_SEEK_END`: the pointer is set to the end of file plus offset

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void
*buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

Reading a File: readFile2.c

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
  MPI_File_read(fh, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Reading a File: readFile2.c

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
  MPI_File_read(fh, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Declaring a File Pointer

Calculating Buffer Size

Opening a File

File seek & Read

Closing a File

Reading a File: readFile1.c

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_at(fh, rank*bufsize, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Single step operation to accomplish the same result as file seek & read but in a thread-safe manner

MPI-I/O API for Writing Files

- While opening the file in the write mode, use the appropriate flag/s in **`MPI_File_open`**: **`MPI_MODE_WRONLY`** Or **`MPI_MODE_RDWR`** and if needed, **`MPI_MODE_CREATE`**
- For writing, use **`MPI_File_set_view`** and **`MPI_File_write`** or **`MPI_File_write_at`**

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
MPI_Datatype etype, MPI_Datatype filetype, char
*datarep, MPI_Info info)
```

```
int MPI_File_write(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)
```

Writing a File: writeFile1.c (1)

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
  int i, rank, size, offset, nints, N=16 ;
  MPI_File fhw;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int buf[N];
  for ( i=0;i<N;i++){
        buf[i] = i ;
  }
...
```

Writing a File: writeFile1.c (2)

```c
offset = rank*(N/size)*sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "datafile",
   MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);

printf("\nRank: %d, Offset: %d\n", rank, offset);

MPI_File_write_at(fhw, offset, buf, (N/size),
   MPI_INT, &status);

MPI_File_close(&fhw);

MPI_Finalize();
return 0;
}
```

Compile & Run the Program on Compute Node

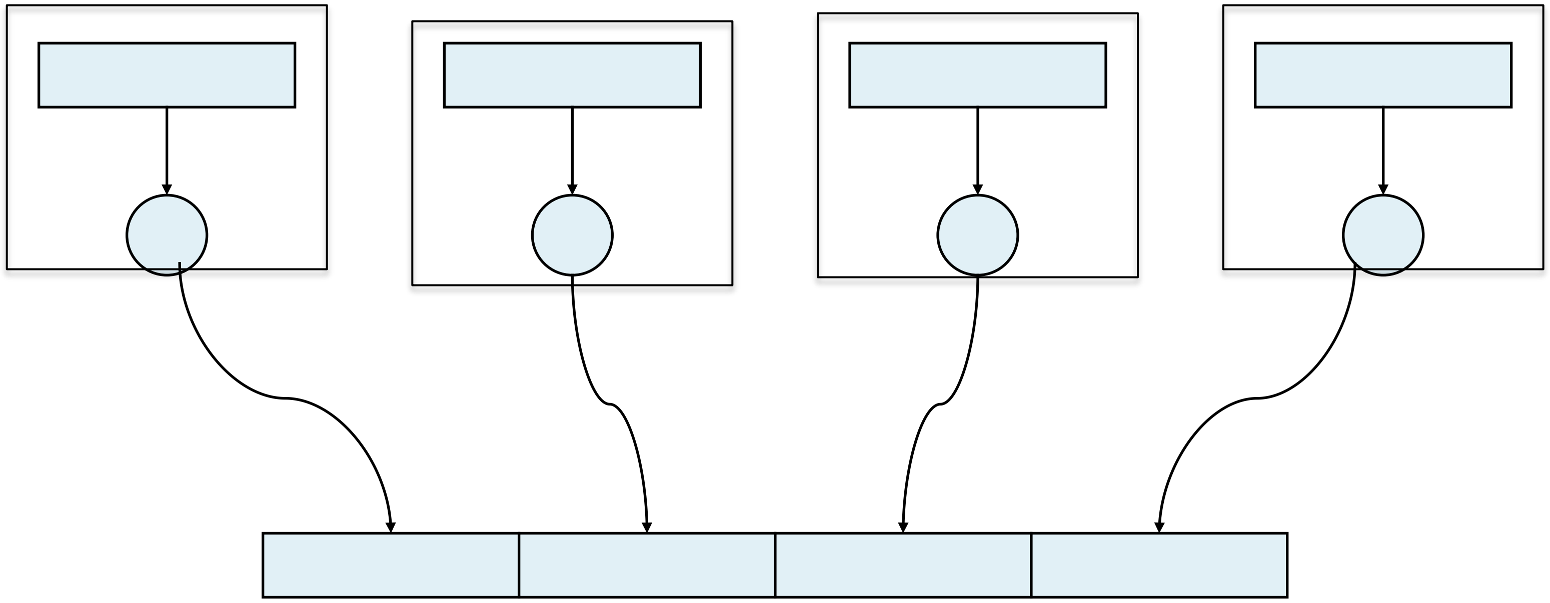
```
c401-204$ mpicc -o writeFile1 writeFile1.c
c401-204$ ibrun -np 4 ./writeFile1
TACC: Starting up job 1754636
TACC: Setting up parallel environment for MVAPICH2+mpispawn.
Rank: 0, Offset: 0
Rank: 1, Offset: 16
Rank: 3, Offset: 48
Rank: 2, Offset: 32

TACC: Shutdown complete. Exiting.
c401-204$ hexdump -v -e '7/4 "%10d "' -e '"\n"' datafile
         0          1          2          3          0          1          2
         3          0          1          2          3          0          1
         2          3
```

File Views for Writing to a Shared File (1)

When processes need to write to a shared file, assign regions of the file to separate processes using **`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)
```



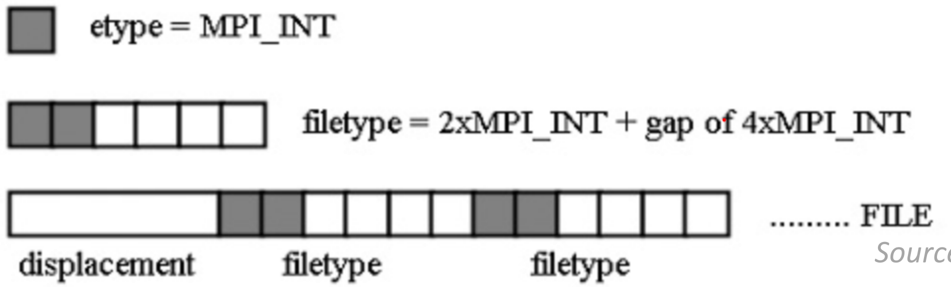
File Views for Writing to a Shared File (2)

- File views are specified using a triplet - (*displacement*, *etype*, and *filetype*) – that is passed to **`MPI_File_set_view`**

  displacement = number of bytes to skip from the start of the file

  etype = unit of data access (can be any basic or derived datatype)

  filetype = specifies which portion of the file is visible to the process



Source: https://www.chpc.utah.edu/images/news/sp2002_Martin07.jpg

- Data representation (datarep above) can be `native`, `internal`, or `external32`

Writing a File: writeFile2.c (1)

```c
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
  int i, rank, size, offset, nints, N=16;
  MPI_File fhw;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int buf[N];
  for ( i=0;i<N;i++){
        buf[i] = i ;
  }
  offset = rank*(N/size)*sizeof(int);
...
```

Writing a File: writeFile2.c (2)

```c
MPI_File_open(MPI_COMM_WORLD, "datafile3",
   MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL,
   &fhw);
  printf("\nRank: %d, Offset: %d\n", rank,
   offset);
  MPI_File_set_view(fhw, offset, MPI_INT,
   MPI_INT, "native", MPI_INFO_NULL);
  MPI_File_write(fhw, buf, (N/size), MPI_INT,
   &status);
  MPI_File_close(&fhw);
  MPI_Finalize();
  return 0;
}
```

Compile & Run the Program on Compute Node

```
c402-302$ mpicc -o writeFile2 writeFile2.c
c402-302$ ibrun -np 4 ./writeFile2
TACC: Starting up job 1755476
TACC: Setting up parallel environment for MVAPICH2+mpispawn.
Rank: 1, Offset: 16
Rank: 2, Offset: 32
Rank: 3, Offset: 48
Rank: 0, Offset: 0

TACC: Shutdown complete. Exiting.
c402-302$ hexdump -v -e '7/4 "%10d "' -e '"\n"' datafile3
         0         1         2         3         0         1         2
         3         0         1         2         3         0         1
         2         3
```

Collective I/O (1)

- Collective I/O is a critical optimization strategy for reading from, and writing to, the parallel file system
- The collective read and write calls force all processes in the communicator to read/write data simultaneously and to wait for each other
- The MPI implementation optimizes the read/write request based on the combined requests of all processes and can merge the requests of different processes for efficiently servicing the requests
- This is particularly effective when the accesses of different processes are noncontiguous

Collective I/O (2)

- The collective functions for reading and writing are:
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`

- Their signature is the same as for the non-collective versions

MPI-I/O Hints

- MPI-IO hints are extra information supplied to the MPI implementation through the following function calls for improving the I/O performance
  - `MPI_File_open`
  - `MPI_File_set_info`
  - `MPI_File_set_view`

- Hints are optional and implementation-dependent
  - you may specify hints but the implementation can ignore them

- `MPI_File_get_info` used to get list of hints, examples of Hints: **`striping_unit, striping_factor`**

Lustre – setting stripe count in MPI Code

- MPI may be built with Lustre support
  - MVAPICH2 & OpenMPI support Lustre

- Set stripe count in MPI code
  Use MPI I/O hints to set Lustre stripe count, stripe size, and # of writers

  Fortran:
  ```
  call mpi_info_set(myinfo, "striping_factor", stripe_count, mpierr)
  call mpi_info_set(myinfo, "striping_unit", stripe_size, mpierr)
  call mpi_info_set(myinfo, "cb_nodes", num_writers, mpierr)
  ```

  C:
  ```
  mpi_info_set(myinfo, "striping_factor",stripe_count);
  mpi_info_set(myinfo, "striping_unit", stripe_size);
  mpi_info_set(myinfo, "cb_nodes", num_writers);
  ```

- Default:
  - # of writers = # Lustre stripes

HDF5 and Parallel HDF5

Hierarchical Data Format (HDF)

- HDF is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of numerical data
- It is portable across operating systems and architectures, and it supports flexible user-defined types
- HDF5 file structure includes two major types of objects:
  - Datasets, which are multidimensional arrays of a homogeneous type
  - Groups, which are container structures which can hold datasets and other groups
- Any HDF5 group or dataset may have an associated attribute list
  - An HDF5 attribute is a user-defined HDF5 structure that provides extra information about an HDF5 object

General Structure of HDF5 Code

```
Open HDF5
Open File
    Open Group
        Open Dataset
            Write Dataset
        Close Dataset
    Close Group
Close File
Close HDF5
```

Code Samples:
https://www.hdfgroup.org/HDF5/examples/intro.html#c

HDF5 Code Example (1)

Source: https://www.hdfgroup.org/ftp/HDF5/current/src/unpacked/examples/h5_crtdat.c

```c
//Example to create a dataset that is a 4 x 6 array
#include "hdf5.h"
#define FILE "dset.h5"
int main() {
   hid_t file_id, dataset_id, dataspace_id;
   //identifiers
   hsize_t dims[2];herr_t  status;
   //Create a new file using default properties
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
   //Create the data space for the dataset
   dims[0] = 4; dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);
```

HDF5 Code Example (2)

Source: https://www.hdfgroup.org/ftp/HDF5/current/src/unpacked/examples/h5_crtdat.c

```
    //Create the dataset
    dataset_id = H5Dcreate2(file_id, "/dset",
  H5T_STD_I32BE, dataspace_id, H5P_DEFAULT, H5P_DEFAULT,
  H5P_DEFAULT);
    //End access to dataset & release resources it uses
    status = H5Dclose(dataset_id);
    //Terminate access to the data space
    status = H5Sclose(dataspace_id);
    //Close the file
    status = H5Fclose(file_id);
}
```

Compiling and Running the HDF5 Code

For Unix platforms, the following compile scripts are included with the binary distribution of the HDF5 software:

h5cc: compile script for HDF5 C programs.

h5fc: compile script for HDF5 F90 programs.

h5c++: compile script for HDF5 C++ programs.

Following are examples of compiling and running an application with the Unix compile scripts:

```
h5fc myprog.f90
./a.out

h5cc -o myprog myprog.c
./myprog
```

Dump of the Output File from the HDF5 Example Program

staff$ h5dump dset.h5

```
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 0, 0, 0, 0, 0, 0,
      (1,0): 0, 0, 0, 0, 0, 0,
      (2,0): 0, 0, 0, 0, 0, 0,
      (3,0): 0, 0, 0, 0, 0, 0
      }
   }
}
}
```

Highly Recommend to Explore Parallel HDF5 on your own ☺

https://www.hdfgroup.org/HDF5/Tutor/parallel.html

https://www.hdfgroup.org/HDF5/PHDF5/

Overview

- Introduction to Data Intensive Computing and application I/O
- I/O During Pre-Processing Stage
  - Choose the right data transfer protocol
  - Selective I/O
- I/O During Processing Stage
  - Understand your parallel file system
  - Use parallel I/O
- **I/O During Post-Processing Stage**
  - Move your data to secondary or tertiary storage media
- An Example of a System for Data Intensive Computing
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

Storage, Archival, and Information Visualization

- After your application has finished running, you might need to move the data involved to secondary or tertiary storage
- Depending upon the analysis that you might be doing, you could be generating multiple data products from your dataset
  - Preserve the raw data and the algorithms to generate the data products for data provenance purposes
    - Note: what might be noise for you could be useful data for someone else
  - As per your need (which should already be explained in the data management plan), you might want to retain different data products for different periods of time

Overview

- Introduction to Data Intensive Computing and application I/O
- I/O During Pre-Processing Stage
  - Choose the right data transfer protocol
  - Selective I/O
- I/O During Processing Stage
  - Understand your parallel file system
  - Use parallel I/O
- I/O During Post-Processing Stage
  - Move your data to secondary or tertiary storage media
- **An Example of a System for Data Intensive Computing**
  - Interesting features: array of about 100,000 NAND flash dies - a 3-D RAID configuration to tolerate failures

Hadoop

- In ~2011, we discovered an exciting new failure mode in large-scale systems
- We called this failure mode "Hadoop"
- Recipe for successful Hadoop failure: take a big system
  - A huge central filesystem
  - Optimized for large, sequential, access
  - With a highly tuned, low-level C interface
- And on that run software that:
  - Assumes a small, massively distributed filesystem
  - Optimized for very small files
  - With an untuned, well… Java
- Results: Deployed Rustler to keep such users off the supercomputers

Wrangler: An XSEDE Resource for Data Intensive Computing

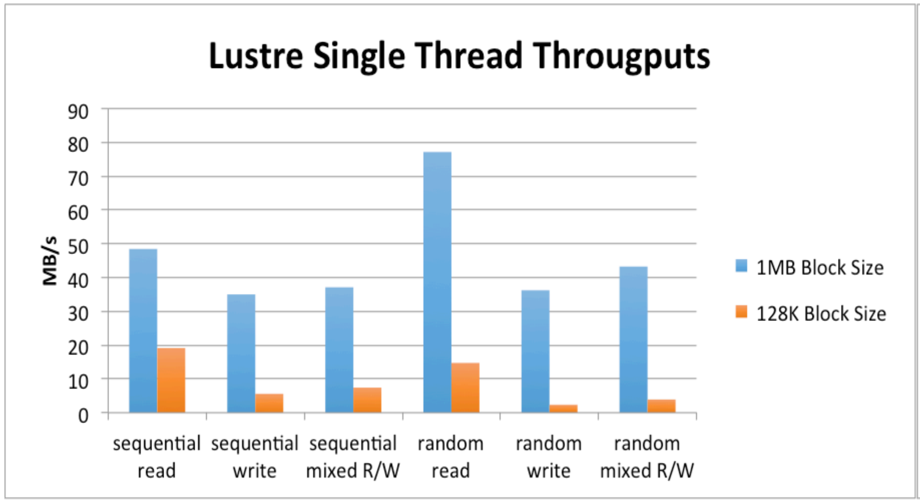
Wrangler provides many different services to help researchers solve their data computing needs, and has

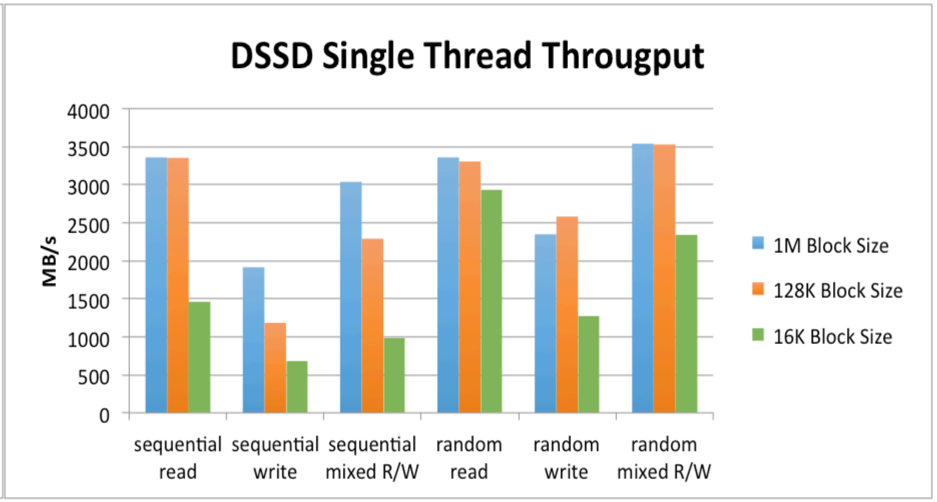
- Massive, replicated, secure high performance data storage (10PB each at Indiana and TACC)
- A large scale flash storage tier for analytics, with bandwidth of 1TB/s and 250M IOPS (6x faster than Stampede)
- Embedded processing of more than 3,000 processors cores for data analysis
- Flexible support for a wide range of data workflows, including those using Hadoop and databases
- Integration with Globus Online services for rapid and reliable data transfer and sharing
- A fully scalable design that can grow with the amount of users and as data applications grow.

DSSD Storage

- The flash storage provides the truly "innovative capability" of Wrangler
  - **Note:** Dell EMC has stopped selling DSSD.
- Not SSD; a custom interface allows access to the NAND flash technology performance without the overhead of the traditional "disk" interface
- Opportunity to explore APIs that integrate natively with apps (*i.e.,* HDFS direct integration)
- Half a petabyte of usable space
- Nearly 100K NAND flash dies
- 960 Gen3 x4 PCI links to the storage system

Where DSSD Really Shines





- Single thread IO for different block sizes
  - Flash is faster than single spinning disk (no surprise)
  - DSSD sustains most throughput for small block sizes and for sequential and random I/O patterns

Wrangler in the TACC Ecosystem

- TACC is traditionally a provider of HPC, Visualization, and storage systems and we still are
- But new communities provide kinds of data-intensive problems our HPC systems just aren't built for
  - Run Hadoop on your favorite supercomputer to see what we need
  - Or do a bunch of random access to a bunch of really small files
- Wrangler is not to replace our supercomputer, visualization, or cloud offerings; it supplements this environment.

References

1. HDF5 Tutorial:
   www.hdfgroup.org/HDF5/Tutor/introductory.html

2. NICS I/O guide:
   http://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips#lustre-fundamentals

3. T3PIO: github.com/TACC/t3pio

4. Introduction to Parallel I/O:
   http://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf

5. Introduction to Parallel I/O and MPI-IO by Rajeev Thakur

6. An Analysis of State-of-the-Art Parallel File Systems for Linux

http://www.linuxclustersinstitute.org/conferences/archive/2004/PDF/20-Margo_M.pdf

Thanks!

Questions or Comments?

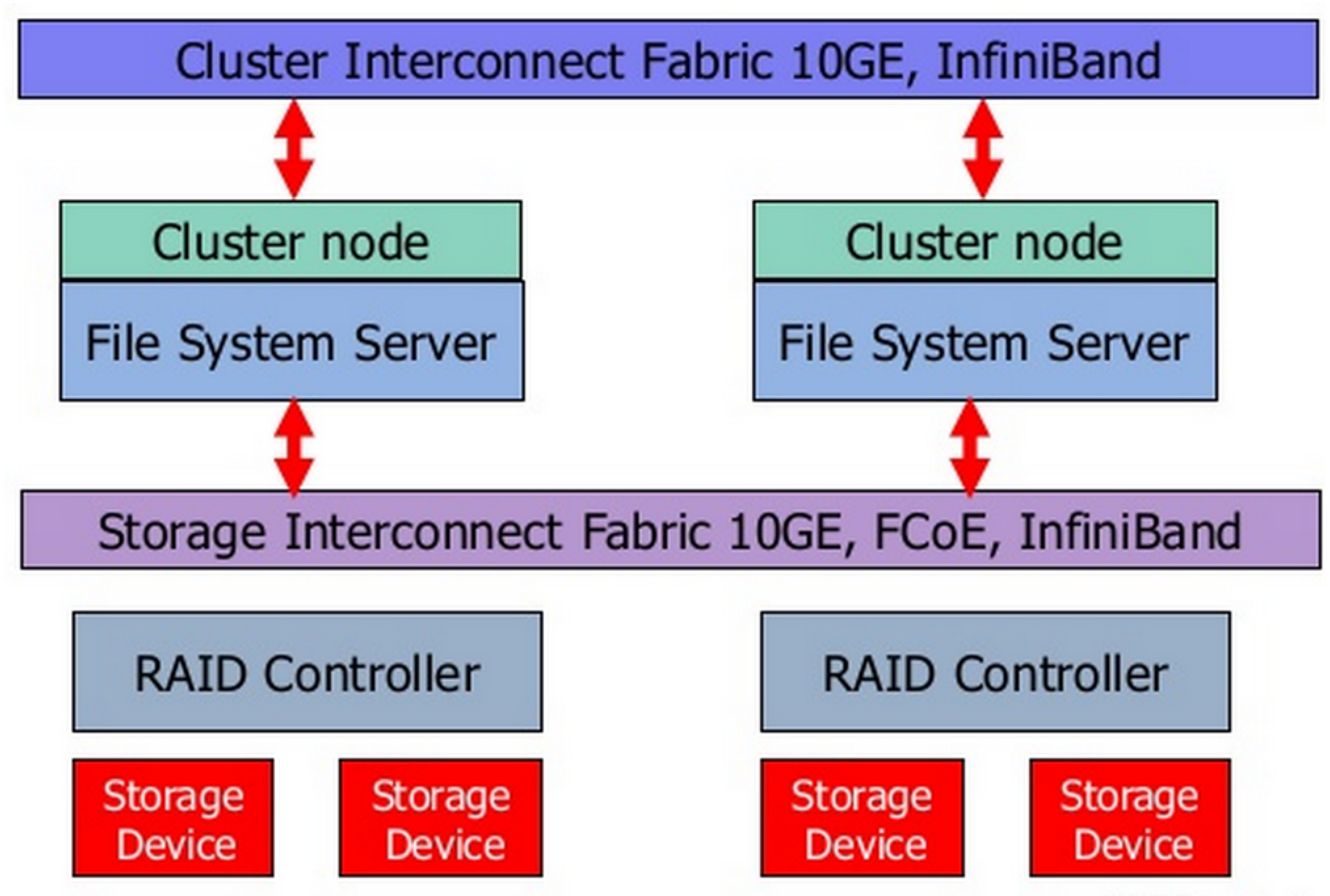
Compute-Intensive Applications Versus Data Intensive Applications

- Compute-Intensive Applications devote most of their execution time to computational requirements

- Data-Intensive Applications devote most of their execution time to I/O and manipulation of data, and read/write large volumes of data, for example:
  - Running simulations for studying climate change over last 10 years
  - eDiscovery

GPFS Topology 1

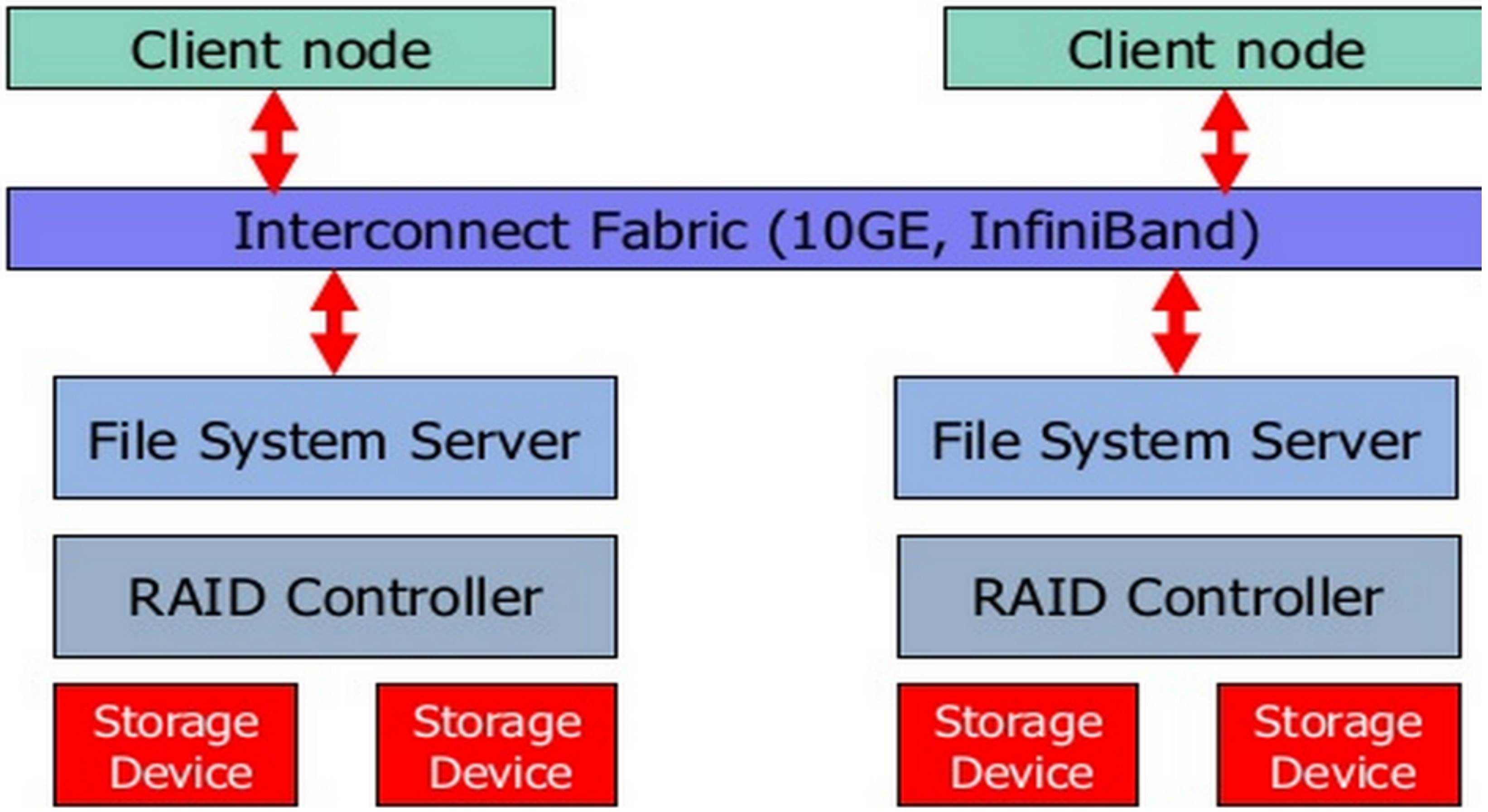
Direct Attached Storage



Source: http://www.slideshare.net/GabrielMateescu/sonas-44390281

GPFS Topology 2

Network Attached Storage



Source: http://www.slideshare.net/GabrielMateescu/sonas-44390281

GPFS versus Lustre

	GPFS	**Lustre**
MDS	In direct-attached storage topology, all nodes acts like MDS, whereas in network-attached topology, some nodes (server nodes) act like MDS	Often 1 primary + 1 failover; since version 2.4, supported for clustered MDS is available
Storage Type	RAID, SAN, …	RAID, SAN, …
User Control for Tuning	None; optimized by administrators at the time of installation	User can change some parameters like stripe size and stripe count
Daemon Communication	TCP/IP	Portal
License	Proprietary (IBM product)	Open-Source

Source: Reference 6

Note about atomicity Read/Write

```
int MPI_File_set_atomicity ( MPI_File mpi_fh, int flag );
```

- Use this API to set the atomicity mode – 1 for true and 0 for false – so that only one process can access the file at a time
- When atomic mode is enabled, MPI-IO will guarantee sequential consistency and this can result in significant performance drop
- This is a collective function

This example creates a parallel HDF5 file (1)

```
#include "hdf5.h"
#define H5FILE_NAME "SDS_row.h5"
int main (int argc, char **argv){
    // HDF5 APIs definitions
    hid_t file_id;
   //file and dataset identifiers
    hid_t plist_id; //property list identifier
    herr_t status;
    //MPI variables
    int mpi_size, mpi_rank;
    MPI_Comm comm  = MPI_COMM_WORLD;
    MPI_Info info  = MPI_INFO_NULL;
    //Initialize MPI
    MPI_Init(&argc, &argv);
```

This example creates a parallel HDF5 file (2)

```
    MPI_Comm_size(comm, &mpi_size);
    MPI_Comm_rank(comm, &mpi_rank);
    //Setup file access property list with parallel I/O acces
     plist_id = H5Pcreate(H5P_FILE_ACCESS);
     H5Pset_fapl_mpio(plist_id, comm, info);
     //Create a new file collectively.
    file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
     //Close property list.
    H5Pclose(plist_id);
    // Close the file.
    H5Fclose(file_id);
    MPI_Finalize();
    return 0;
}
```