TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Parallel I/O

IHPCSS

July 9, 2024

PRESENTED BY:

John Cazes

cazes@tacc.utexas.edu

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Example: Write a 2D distributed array in parallel
- Introduction to HDF5
- I/O Strategies

I/O in HPC Applications

- High Performance Computing (HPC) applications often
  - Read initial conditions or datasets for processing
  - Write numerical data from simulations
    - Saving application-level checkpoints
    - More writes than reads

- In case of large distributed HPC applications, the total execution time 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

Addressing the I/O Bottlenecks

- Software support for parallel I/O is available in the form of
  - Parallel distributed file systems that provide parallel data paths to storage disks
  - MPI I/O
  - Libraries like HDF5, NetCDF, PnetCDF

- Understand the I/O strategies for maintaining good citizenship on a supercomputing resource

Some Examples of HPC Parallel File Systems

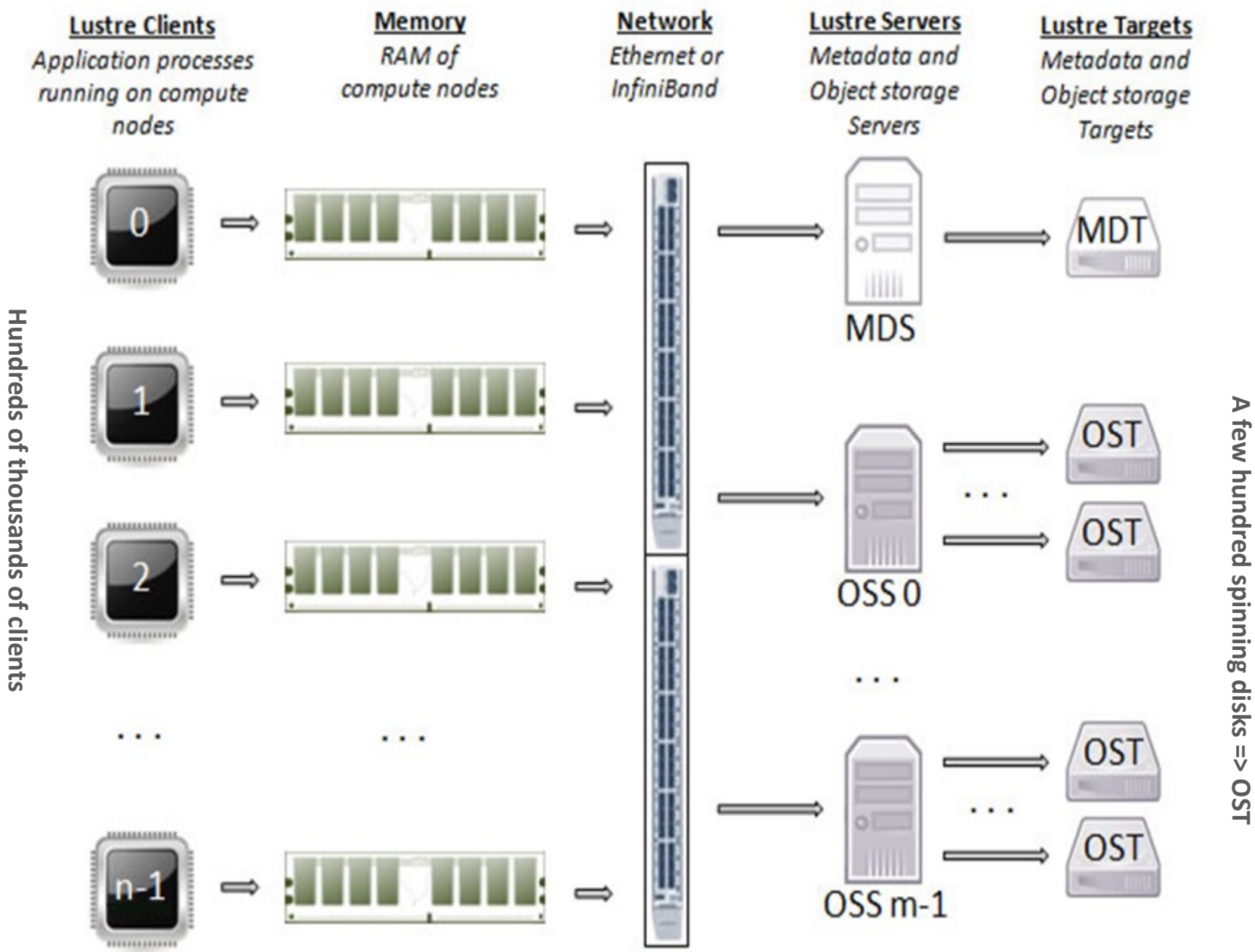
- Lustre File System
- BeeGFS
- IBM Storage Scale (GPFS)
- VAST
- WEKA

Lustre File System

Lustre Components
- Clients -- compute nodes
- Metadata – MDS
- Object Storage Servers – OSSs
- Object Storage Targets – OSTs

Access from clients coordinated by MDS

Data stored on OSTs



Source: https://oit.utk.edu/hpsc/isaac-open/lustre-user-guide/

Lustre Filesystems

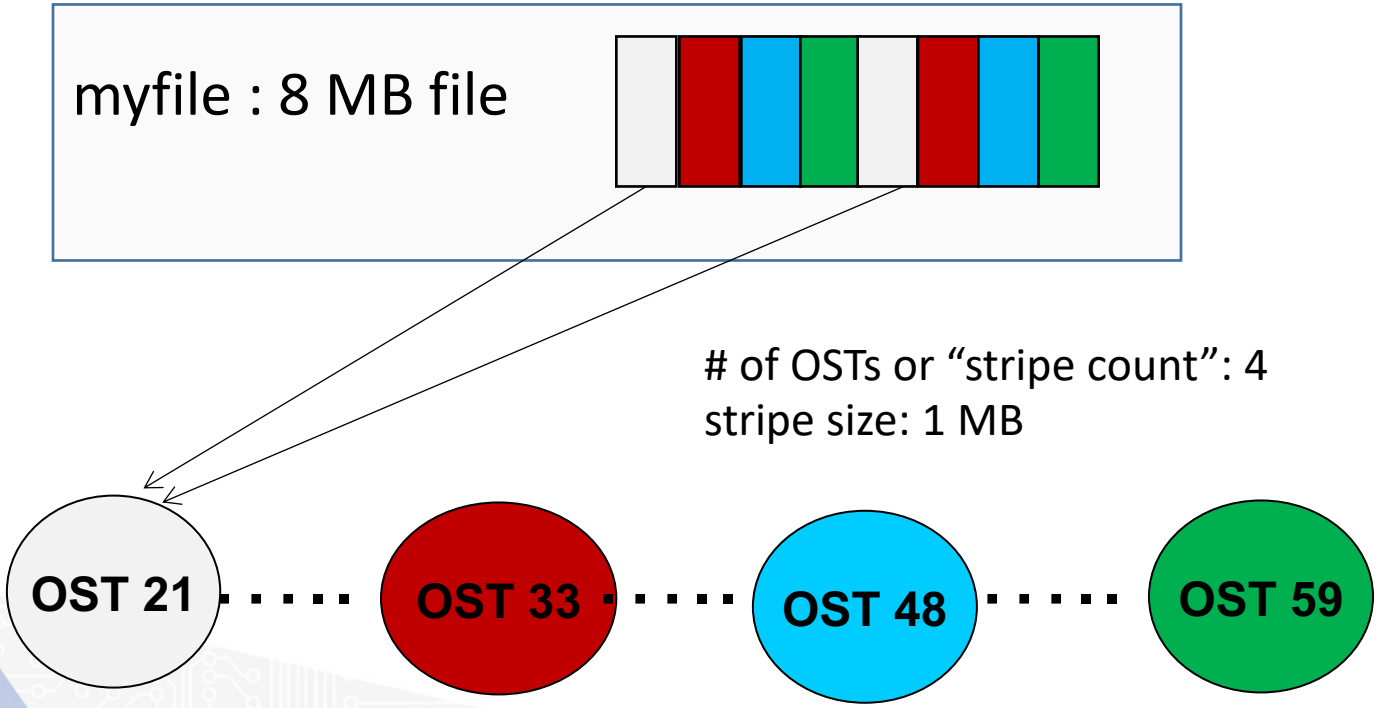
- Each Lustre filesystem has a different number of OSTs
- The greater the number of OSTs the greater the bandwidth
- To check the number of OSTs available on the filesystems, you may use the command:

```
$ lfs osts
```

	$HOME	$PROJECT/$WORK	$SCRATCH
Bridges2	12	24	
Frontera	4	24	16/16/32

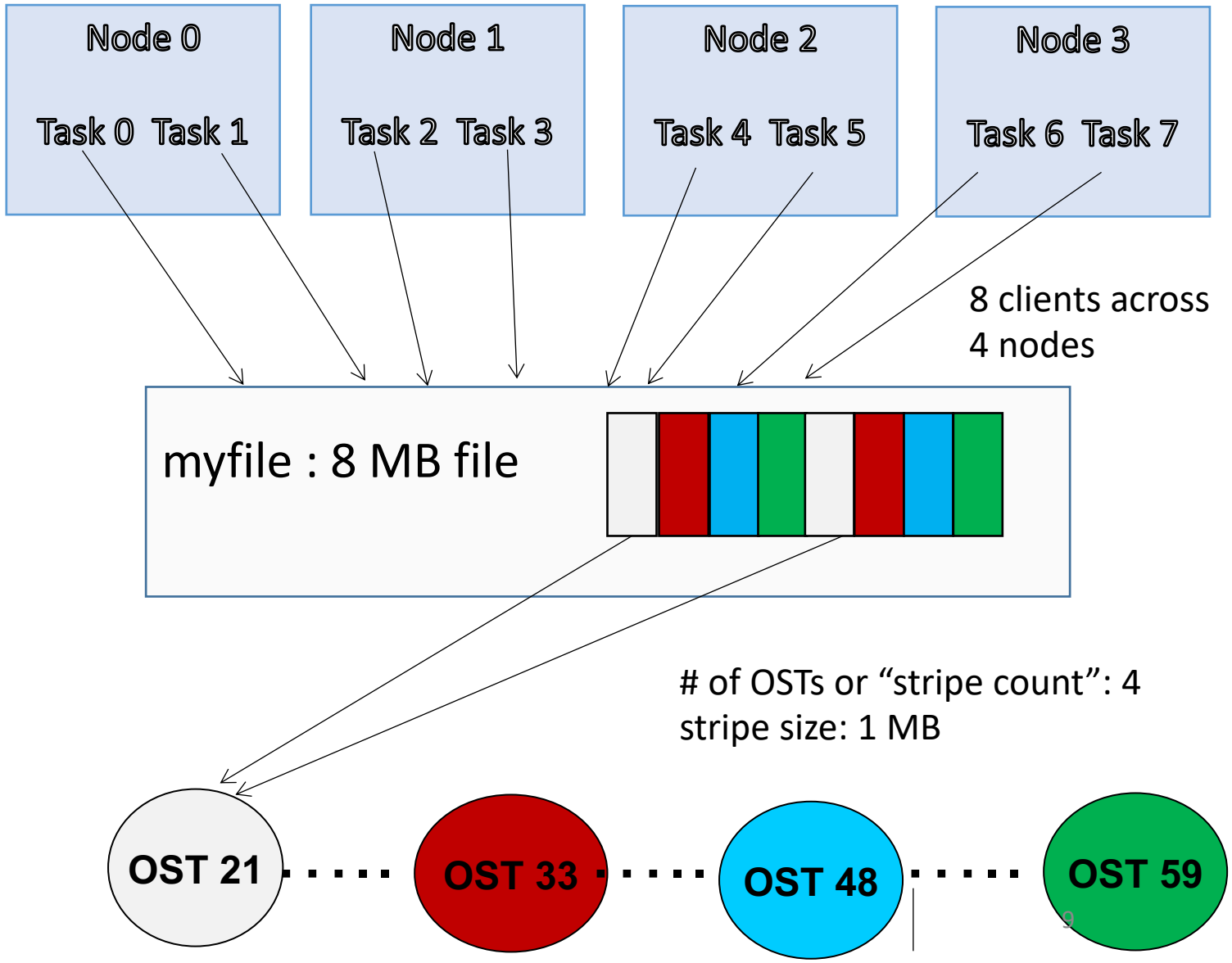
Lustre File System - Striping

- Lustre supports the striping of files across several I/O servers (similar to RAID 0)
- Each stripe is a fixed size block
- Multiple stripes translates to parallel I/O on the back end



Lustre File System – Multiple clients

- Lustre supports access by multiple clients
- read/write operations from multiple clients translates to parallel I/O on the front end
- Each metadata operation open/close/seek/stat is serial



Lustre File System – Striping

Administrators set a default stripe count and stripe size that applies to all newly created files

Bridges: `$PROJECT`: 1 stripes/1MB

Frontera: `$SCRATCH`: 1 stripes/1MB

This is best for multiple concurrent I/O operations to multiple files.

Multiple stripe counts improves bandwidth for I/O from multiple clients to a single file.

Users can reset the default stripe count or stripe size using Lustre commands

Lustre Commands

Get stripe count

```
% lfs getstripe ./testfile
./testfile
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  31
        obdidx           objid          objid            group
            31         6087301       0x5ce285                0
```

Set stripe count

```
% lfs setstripe -c 4 -s 4M testfile2
% lfs getstripe ./testfile2
./testfile2
lmm_stripe_count:   4
lmm_stripe_size:    4194304
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  2
        obdidx           objid           objid            group
             2        42306284      0x2858aec                0
            16        42303585      0x2858061                0
            40        42323070      0x285cc7e                0
            42        42317764      0x285b7c4                0
```

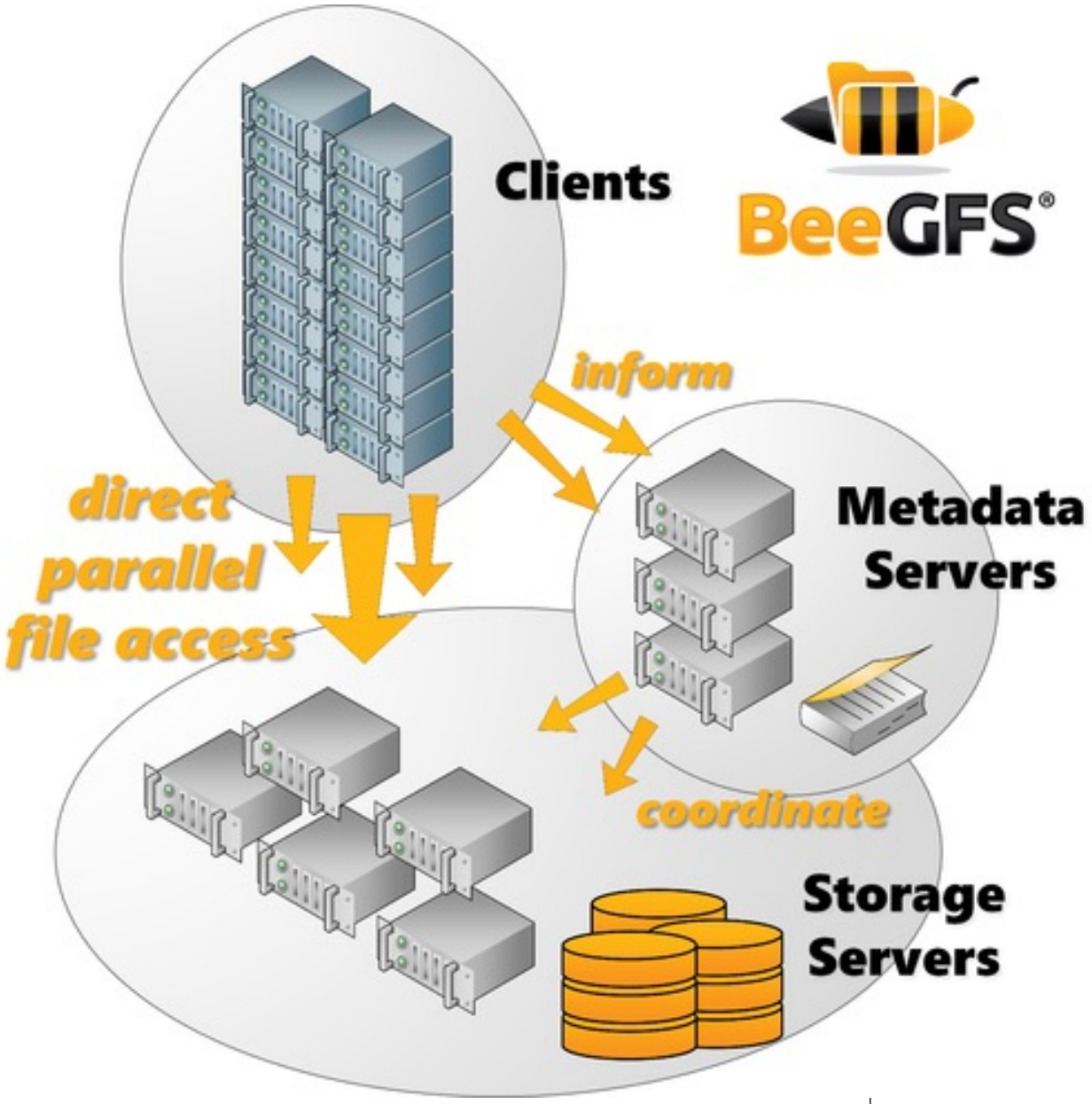
Real-World Scenario
FLASH code: impact of file striping on I/O

LFS Stripe Count #	Time taken for reading a checkpoint file (in seconds)	Time Taken for Writing a Checkpoint file (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

BeeGFS File System

BeeGFS is very similar to Lustre
- Clients
- Storage Servers
- Metadata Servers

Provides a similar level of user control
- Uses chunks instead of stripes
- Can set chunk size
- Unlike lustre you can specify which storage servers to use



Source: https://www.beegfs.io/content/

BeeGFS Commands

Get chunk info

```
% beegfs-ctl --getentryinfo ./testfile
Entry type: file
EntryID: 0-62A1021C-5
Metadata node: metaA-numa1-2 [ID: 5]
Stripe pattern details:
+ Type: RAID0
+ Chunksize: 512K
+ Number of storage targets: desired: 1; actual: 1
+ Storage targets:
  + 92 @ storageF [ID: 6]
```

Set chunk count (only works on directories)

```
% beegfs-ctl --setpattern --numtargets=4 --chunksize=4m ./
New chunksize: 4194304
New number of storage targets: 4
% beegfs-ctl --getentryinfo ./testfile2
Entry type: file
EntryID: 9-62A1025B-5
Metadata node: metaA-numa1-2 [ID: 5]
Stripe pattern details:
+ Type: RAID0
+ Chunksize: 4M
+
```

Lustre Commands

Get stripe info

```
% lfs getstripe ./testfile
./testfile
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  31
        obdidx           objid
            31         6087301
```

Set stripe count

```
% lfs setstripe -c 4 -s 4M testfile2
% lfs getstripe ./testfile2
./testfile2
lmm_stripe_count:   4
lmm_stripe_size:    4194304
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  2
        obdidx           objid
             2        42306284
            16        42303585
```

You Can Stress Out Lustre Easily if You…

Open and close the same file every few milliseconds

- Stresses the MDS

Many Python workflows exhibit this behavior

Conda environment at large scale is especially difficult

Too often, too many

- Stresses the MDS and OSTs

Write large files to `$HOME`

- `$SCRATCH` should be used

`ls` in an overfull 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 listing is serial
- Wildcard expansion breaks for huge file counts

What happens when Lustre gets stressed out?

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, HDF5, NetCDF

Applications, *e.g.*, FLASH, WRF, OpenFOAM
IO Libraries, *e.g.,* HDF5, NetCDF, PnetCDF
Parallel I/O libraries, *e.g.,* MPI-I/O
Parallel File Systems, *e.g.,* Lustre, BeeGFS, Spectrum Storage
Data stored on Disk

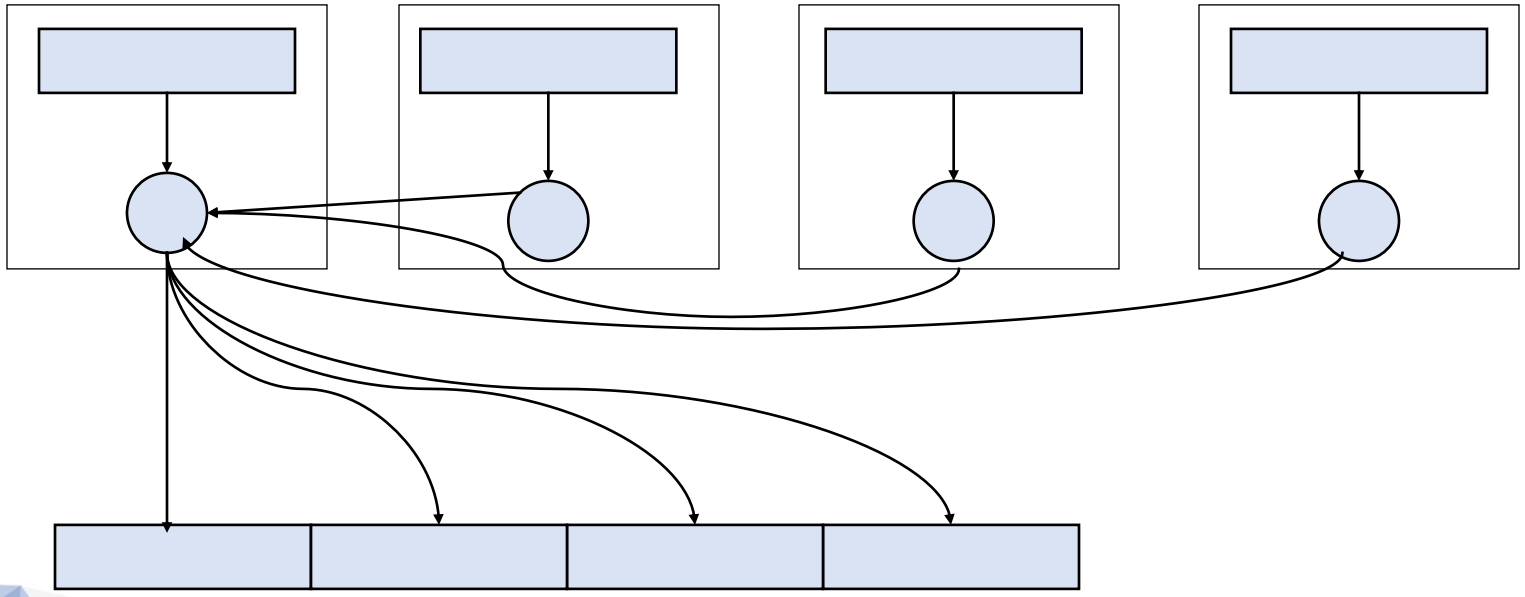
Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Example: Write a 2D distributed array in parallel
- Introduction to HDF5
- I/O Strategies

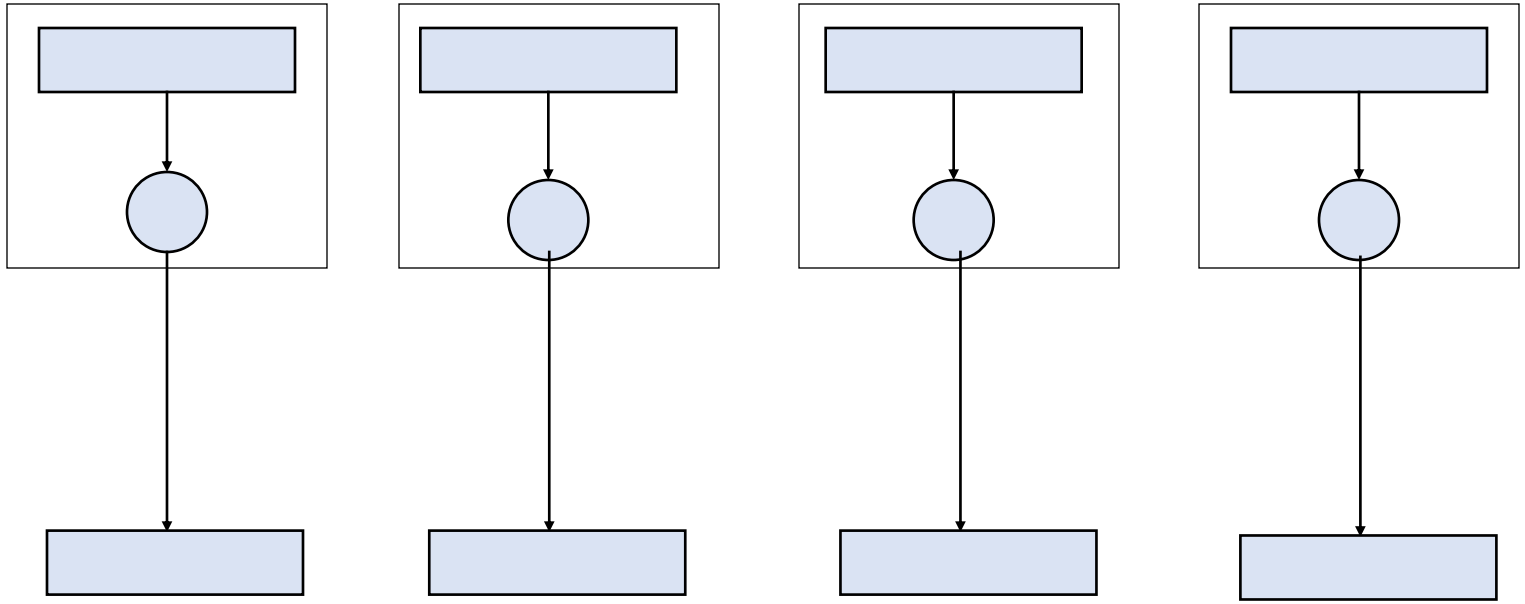
Typical Pattern:
Parallel Programs Doing Sequential I/O

- 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

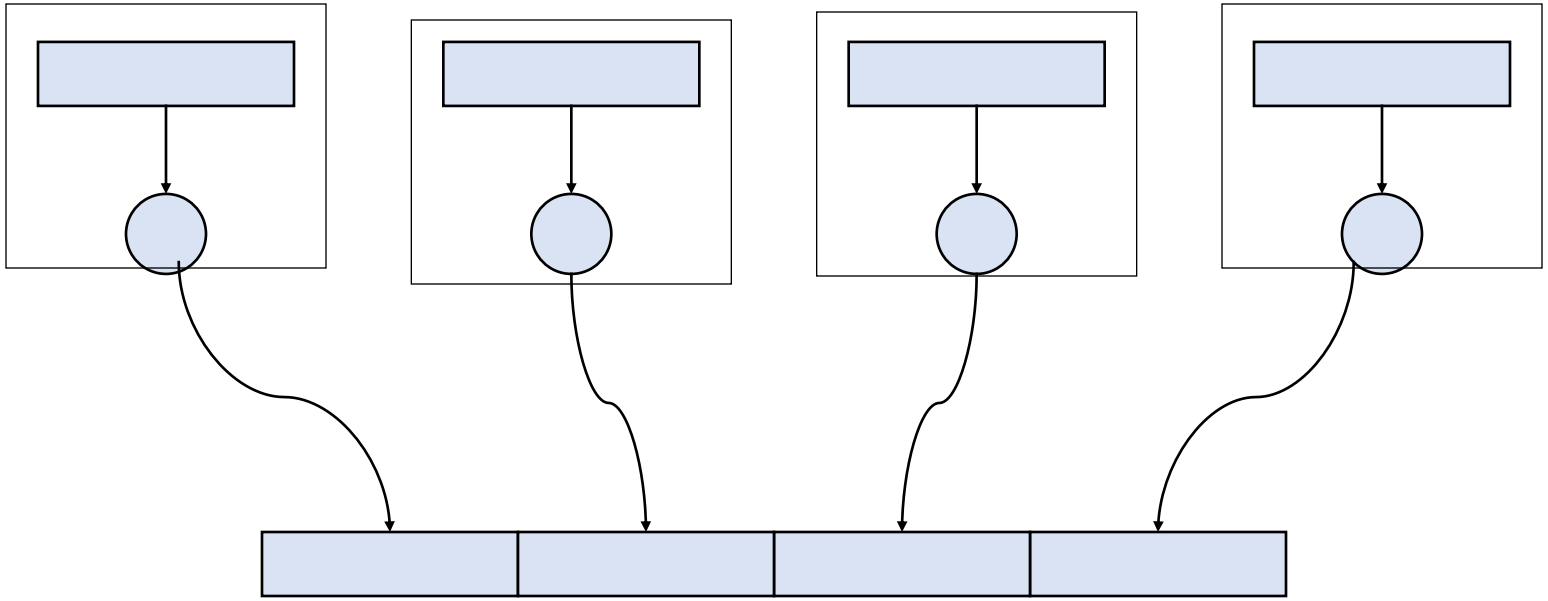


Another Pattern:
Each Process Writing to a Separate File



Useful 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 provides a single file for storage and transfer purposes



Outline

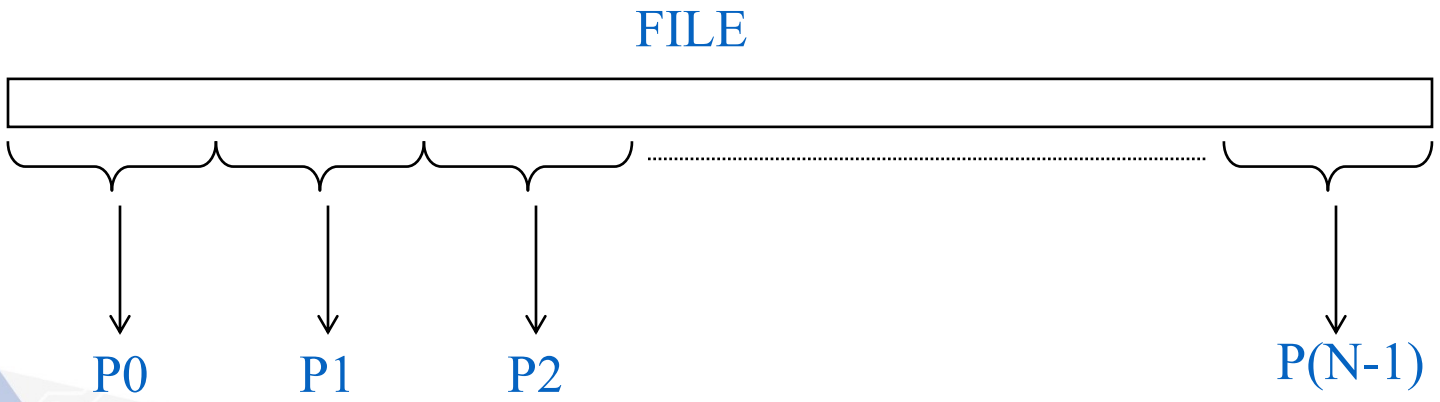
- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Example: Write a 2D distributed array in parallel
- Introduction to HDF5
- I/O Strategies

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 file system 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`/`MPI_File_write`)
  - Calculate byte offsets (*e.g.*, `MPI_File_read_at`/`MPI_file_write_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 mode, 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)
```

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)
```

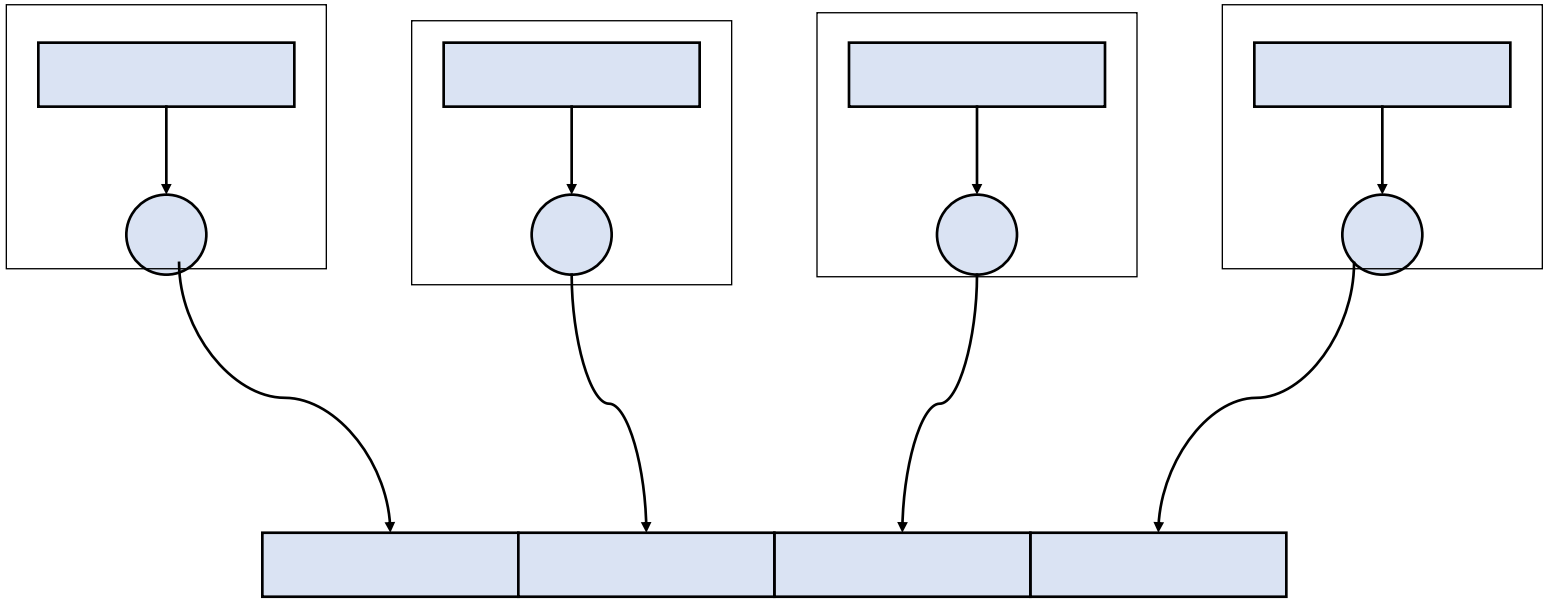
```
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)
```

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

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)
```



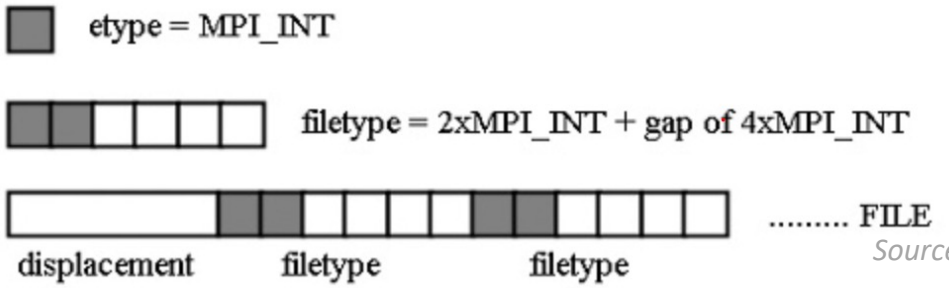
File Views for Writing to a Shared File (2)

- 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 on previous slide) can be `native`, `external32`, or user defined

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

MPI-I/O Optimization

Lustre aware MPI-I/O on a Lustre file system

Collective I/O

- Set stripe count to number of nodes
- Set stripe count to 75% of number of OSTs

Non-collective I/O

- Set stripe count to number of nodes **IF** all tasks writing to a single file
- Set stripe count to one **IF** each task is writing to its own file
- Use hints if creating the file in application
- Use `lfs setstripe command` to set default for output file or directory

- Use hints in `MPI_Info_set` if creating the file in application
- Use `lfs setstripe command` to set default for output file or directory

Other file system or not Lustre enabled

Collective I/O

- Set cb_nodes to the number of nodes using `MPI_Info_set` – this sets the number of writers

Outline

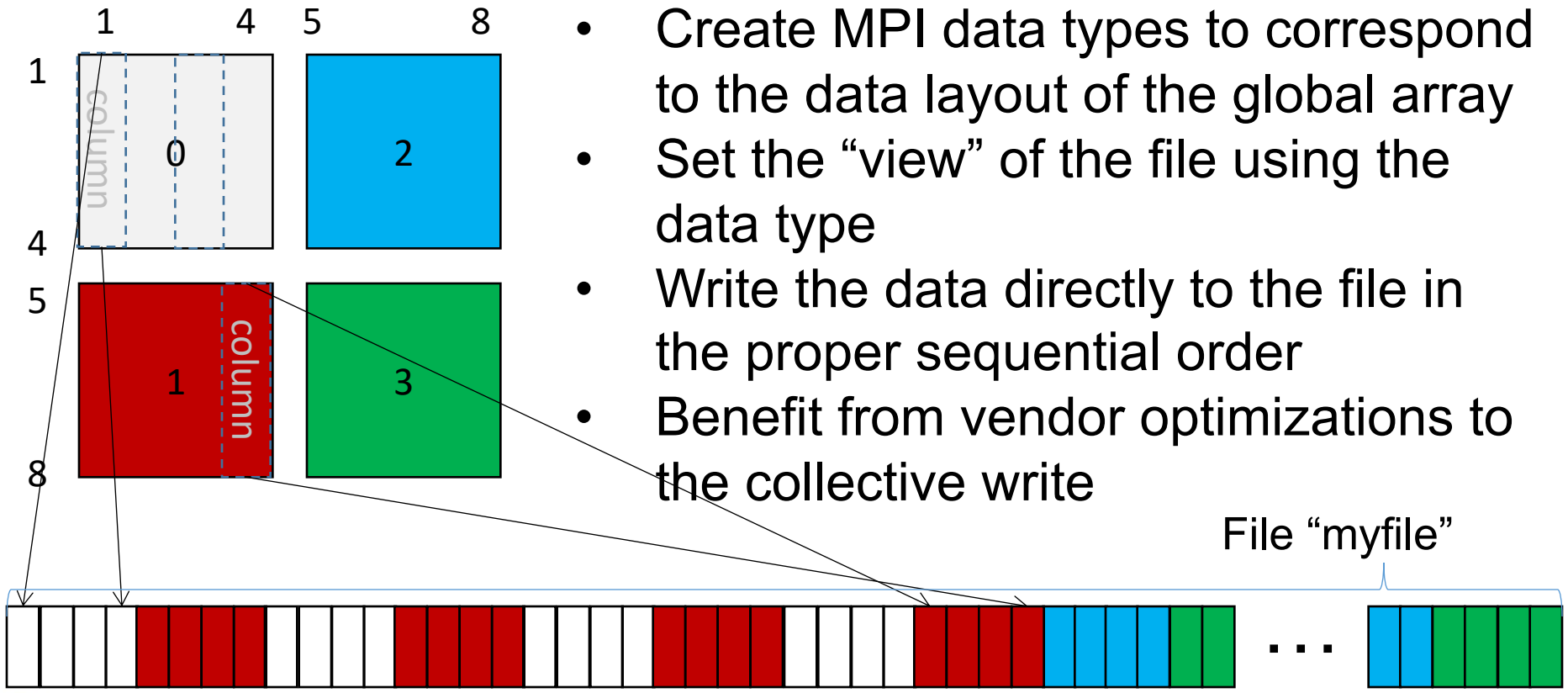
- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Example: Write a 2D distributed array in parallel
- Introduction to HDF5
- I/O Strategies

Example: 2D distributed array

- Write a 2D distributed array in parallel to a single file
- Use collectives to optimize write
- Data in the file is stored in sequential order

MPI Datatypes

Use MPI datatypes and a “view” on a file to map local data to a single file

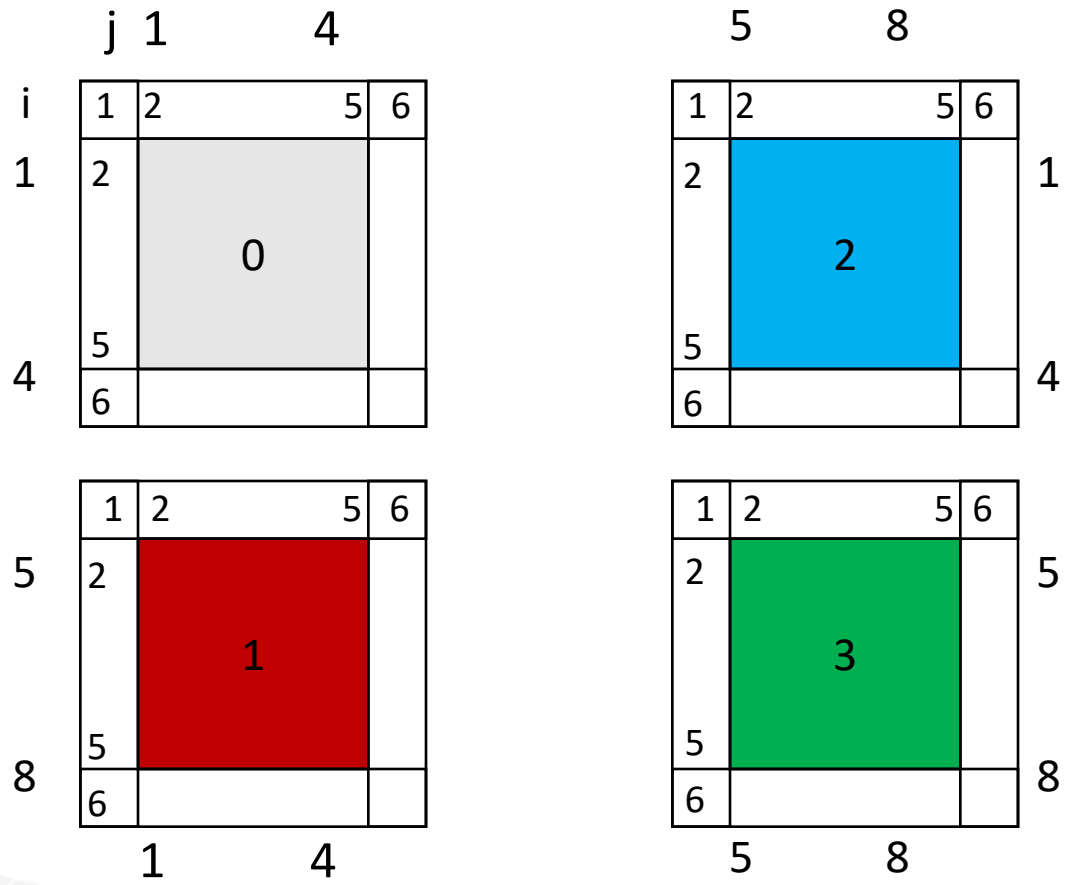


- Create MPI data types to correspond to the data layout of the global array
- Set the “view” of the file using the data type
- Write the data directly to the file in the proper sequential order
- Benefit from vendor optimizations to the collective write

Data Layout

Example: Write an 8x8 data array with a halo to a file in sequential order

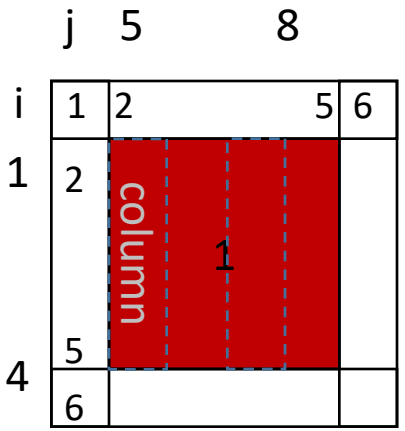
```
allocate(data1(6,6) !4x4 local data array with halo
```



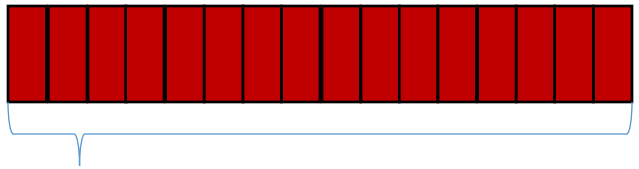
Datatype for Local Map

Create an MPI datatype to map the local data

(MPI uses C style indexing. So, indices provided to MPI start from 0 .)




```
size(1) = 6; s_size(1)=4; start(1)=1 !local i index
size(2) = 6; s_size(2)=4; start(2)=1 !local j index
call mpi_type_create_subarray(2,size,s_size,start, &
   MPI_ORDER_FORTRAN,MPI_REAL8,local_data,mpierr)
call mpi_type_commit(local_data,mpierr)
```

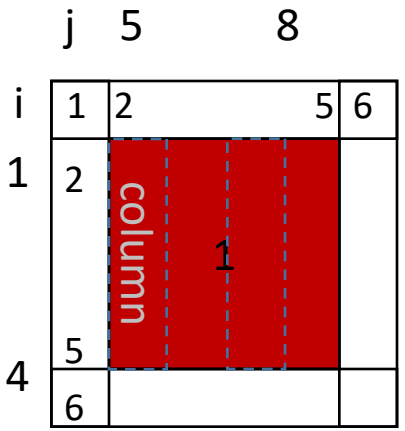


Local mapping of data

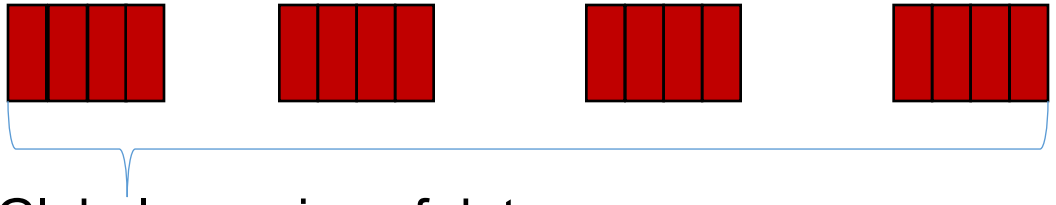
Datatype for Global Map

Create an MPI datatype to map the global data

(MPI uses C style indexing. So, indices provided to MPI start from 0 .)



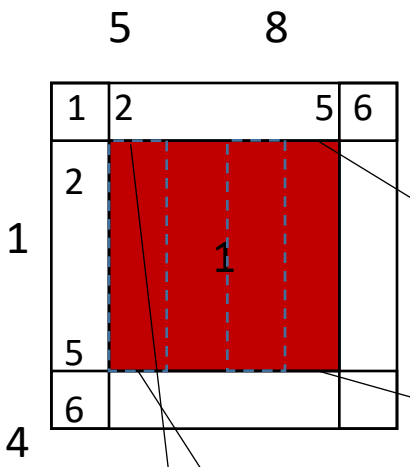

```
size(1) = 8; s_size(1)=4; start(1)=0 !global i index
size(2) = 8; s_size(2)=4; start(2)=4 !global j index
call mpi_type_create_subarray(2,size,s_size,start, &
   MPI_ORDER_FORTRAN,MPI_REAL8,global_data,mpierr)
call mpi_type_commit(global_data,mpierr)
```



Write the File

Open the file

```
call mpi_file_open(MPI_COMM_WORLD,"myfile",MPI_MODE_CREATE, &
     MPI_INFO_NULL,filehandle,mpierr)
```

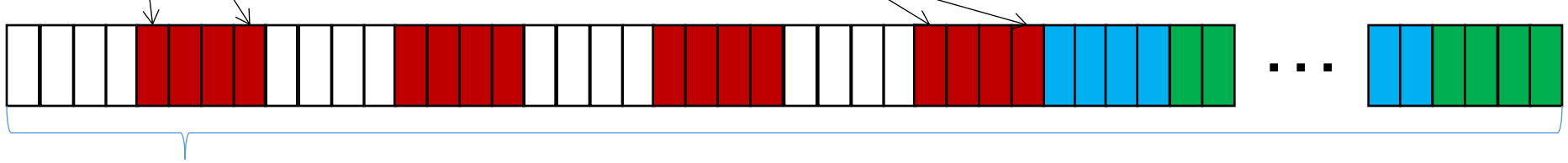


Set the view

```
file_offset=0
call mpi_file_set_view(filehandle,file_offset,&
  MPI_REAL8,global_data,"native",MPI_INFO_NULL,mpierr)
```

Write the file

```
call mpi_file_write_all(filehandle,data1,1, &
  local_data,mpistat,mpierr)
```



File "myfile" is arranged as if the data array was written sequentially

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Example: Write a 2D distributed array in parallel
- Introduction to HDF5
- I/O Strategies

HDF5: Hierarchical Data Format

HDF5 is a file format

- It may be used to manage any kind of data.
- An HDF5 file can be viewed as a file system inside a file.
- It uses a Unix style directory structure.
- It is a mixture of entities: groups, datasets, and attributes.
- Any entity can have descriptive attributes (metadata), e.g. physical units.

HDF5 Nice Features

- Interface support for C, C++, Fortran, Java, and Python
- Supported by data analysis packages
  (Matlab, IDL, Mathematica, Octave, Visit, Paraview, Tekplot, etc.)
- Machine independent data storage format
- Supports user defined datatypes and metadata
- Read or write to a portion of a dataset (Hyperslab)
- Runs on almost all systems

HDF5: The Benefits of Metadata

- It is easy to record many metadata items within a solution file.

- Adding attributes later won't break any program that reads the data.

- With HDF5 it is easy to save with each solution file:
  - Computer Name, OS Version.
  - Compiler and MPI name and version.
  - Program Version.
  - Input file.
  - …

Parallel HDF5 Overview

- Parallel HDF5 library
  - You can write one file in parallel efficiently!
  - Parallel performance of HDF5 very close to MPI I/O.
- Uses MPI I/O (Don’t reinvent the wheel)
- MPI I/O techniques apply to HDF5.
- Use MPI_Info object to control # writers, # stripes(Lustre), stripe size(Lustre), etc.

Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O Pattern
- Introduction to MPI I/O
- Example: Write a 2D distributed array in parallel
- Introduction to HDF5
- I/O Strategies

General Strategies for I/O

- Access data contiguously in memory and on disk if possible
- Avoid “Too often, too many” access pattern
- Write one global file instead of multiple files
- Use parallel I/O
  - MPI I/O
  - Parallel HDF5, parallel NetCDF
- Set file attributes (stripe count, stripe size, #writers) properly

Summary

I/O can impact performance at large scale

- Take advantage of the parallel file system
- Consider using MPI-IO, Parallel HDF5, or Parallel NetCDF libraries
- Analyze your code to determine if you may benefit from parallel I/O
- Set stripe count and stripe size for optimal use if on a Lustre file system

References

1. HDF5 Tutorial:
   https://portal.hdfgroup.org/hdf5/develop/_intro_par_h_d_f5.html

2. UTK I/O guide:
   https://oit.utk.edu/hpsc/isaac-open/lustre-user-guide/

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

4. MPI – The complete Reference: Volume 2, The MPI Extensions

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