Data Transfer, Data Archiving, and Parallel I/O

EU-US Summer School on
High Performance Computing

New York, NY, USA

June 27, 2013

Lars Koesterke: Research Staff @ TACC

Outline

- Motivation
- Data transfer
  - `scp` and beyond
- Data archiving
  - Tape-based archiving
- Parallel I/O
  - MPI I/O
  - Libraries: HDF5, NetCDF, etc.

- **15 minutes per topic is rather short**
- **I will focus on the why and what, and not on the syntax**
- **The “how-to” is not rocket science, but “why to do what” is not always obvious**

Guts and Glory

- There is no real glory in:
  "Data Transfer, Data Archiving, and Parallel I/O"
- However, getting this wrong may cause a lot of havoc on a cluster
- Getting this wrong will get you very(!) quickly on the "black" list of the system administrator

This is an example of what we send to "naughty" users:

"Your job has caused a failure of the parallel file system. Your account will remain locked until the issues with your application are resolved …"

Main Goal

- To read/write, transfer, and archive files while
  1. not crashing the system
  2. not impeding other users
  3. in a timely manner
- This requires the right tools and patience and a (basic) understanding of the hardware

Operational Details

- The parallel file system is (by far) the most fragile part of a cluster
- Users/application can very easily bring the file system to a collapse
  - Login nodes loose connection to the file system
  - Mount points will get lost
  - The file system may become unresponsive on the compute node and jobs will fail
  - Other users, other jobs are immediately impaired
- The sys-admin constantly monitors the load on the file system and will take action

Remarks

This talk reflects my biased view on the topic

(You may draw different conclusions)

What to do/Best practises!

Why to do what?

What can go wrong?

Motivation

- Dealing with big data (sets) is quite cumbersome
- Why is that?
  - Compute capabilities have exploded (2x every 18 months)
  - Network and I/O bandwidth has not
- Use the right tools, apply the right level of complexity for the job
- Where do the big data sets come from?
  - Traditional fields of HPC: output/snapshots of simulations
  - New areas: input data/data mining

Part I: File Transfer

- Avoid data transfer, if possible
  - Many resources (all clusters at TACC) provide multiple capabilities
    - HPC
    - Visualization nodes with GPU's
    - Special I/O subsystems (Hadoop, etc.)
  - Do everything on one resource, transfer the final result
- Baseline: How much data can you transfer?
  - From my workstation at TACC to Stampede:
    - 10 Mbyte/s → 1 TByte a day
  - 10 GigE connection between XSEDE resources
    - 1GB/s → 100x higher transfer rate (that is shared)

Limitations of File Transfer

- If your data is too large you may have to send hard drives via mail
- Some datasets are even too large for that

Data Transfer: `scp`

- Easy to use
- Safe: You'll notice when the transfer has failed
- Low performance (it is actually not too bad)
- No restart capabilities (you'll have to start all over again)

- `scp`: secure copy
  - Authentication (password) is encrypted
  - Data is encrypted
  - Data encryption is not needed unless you work with classified (think DoD/DoE) or NIH (think patient) data

Data Transfer: `scp`

- When to use `scp`?
  - Repeated transfer of small file(s)
  - One time transfer of medium size file(s)
  - I'd rather transfer data over night than install (and use) a more complex tool on my laptop
- What can go wrong?
  - Not that much
  - You'll be able to clearly see if the transfer was interrupted
  - Comparing the size on source and destination will tell you, if the transfer was successful

Data Transfer: **`sftp`**

- **`sftp`**: secure file transfer (protocol)
  - Authentication is encrypted
  - The data is not encrypted
- **`sftp`** gives you a small performance boost
  - Encryption of data is not very compute intensive anymore; today’s CPUs have become very powerful
  - Data overhead for the encryption is not very large

It is probably just me …

- I want to make sure that
  - all files make it to the destination
  - no files are corrupted
- I always check data integrity after the transfer
  - Extend of checking varies with tools/hardware that I use

Data Transfer: How to keep track of multiple files

- Scenario: You want to transfer a directory with subdirectories and hundreds of files
- How can you check the sizes of all files?
- Answer: you don't (I don't)
- Instead:
  - Create a tar file at the source
  - Transfer single tar file
  - Compare the size on source and destination
  - Unpack on destination, and delete on source

Data Transfer: `rsync`

- Consider `rsync` if you transfer multiple files or a whole directory tree
- `rsync` only transfers files that are either non-existent on the destination, or files that differ
  - For each file a checksum is created on host and destination
  - Only files with a different checksum are transferred
  - This comes handy when a transfer has stalled. The second attempt does not repeat the transfer of files that have been (partially) transferred in the first attempt
- `rsync` also comes handy when creating a backup on USB drive(s)

Data Transfer: `hpn-scp`

- **`hpn-scp`** gives you control over the block size employed during the transfer
- The default block size used by scp may not be optimal for your specific source-destination pair
- I have not used **`hpn-scp`** myself, but I'm told that in can improve performance at times
- http://www.psc.edu/index.php/hpn-ssh

Data Transfer: Multiple Streams

- **`grid-ftp, bbcp, globus`** use multiple streams
  - Think of transferring multiple files at the same time
  - Multiple streams transfer a single file in “parallel”
  - Special data-mover node will be used (not the login node)
- You’ll have to install software on your laptop/ desktop
  - But on your laptop/desktop other limitations are at play
  - Performance may not improve
- Software may already be installed on clusters
  - File transfer between clusters will certainly benefit from multiple streams

Data Transfer: Multiple Streams

- What can go wrong?
- Unfortunately quite a lot!
  - Simply comparing the sizes is not enough anymore
- The different streams transfer different parts of the file. Example with 4 streams:
  - 1st stream → 1st quarter
  - 2nd stream → 2nd quarter
  - etc.
- If the 3rd stream fails the file size may be still OK, but the 3rd quarter of the file will contain octal zeros

Data Transfer: Multiple Streams

- How do you check for file integrity?
- Checking the size is not enough
  - File size → Checksum
- A checksum is calculated from all the bits in a file
- **`md5sum`** is readily available on Linux systems
  - **`md5sum <file>`**
  - Creates a long string (30+ characters) containing letters and numbers
- Using a checksum emphasizes even more the need to transfer one (a few) tar file(s)

TACC

Procedure to Transfer File(s)

- Create tar file(s) on source
- Create checksum
- Transfer file(s)
- Create checksum on destination
- Delete file(s) on source

What not to do!

- The file transfer is
  - either handled by the login nodes (scp, sftp, etc.)
  - or handled by special data movers (gridftp, globus)
- Nodes and file system are shared resources
- Any problem you cause will affect all users and potentially impact running jobs
- Play nice and run only 2 (possibly 4) data transfers at a time
- The load on the front-end and on the file system is constantly (day and night) monitored by the sys-admins

Part II: Long-Term Data Storage

- Storing data on tape is (and probably will be for a while) cheaper than storing data on spinning disk
  - Factor is currently about 5x
- I use Ranch @ TACC as an example in this talk, but other archiving systems work similar
- Storing data on tape comes with its own set of problems

Ranch Architecture

- Front-end: Server + File system (spinning disk)
  - The user interacts with this part
- Back-end: Tape system (and some spinning disks)
  - The back-end is mostly invisible to the user

How does it work?

- Transferring files to Ranch
  - Files are transferred to the front-end
  - Data is stored on spinning disk
  - Later (typically within 12-24 hours) data migrates automatically to tape
- Retrieving files from Ranch
  - Files must first be staged, i.e. moved from tape to spinning disk
  - Once the data has been staged, the file transfer can begin
  - Only data on spinning disk is accessible to the user

What can go wrong? Unfortunately a lot!

- User error
  - Directory listings of files on tape are available
  - You can logon to Ranch and execute `ls –l`
  - However, the file sizes of files on tape will be zero
  - The `sls` command shows you the proper information of a file, including if it is on tap or on spinning disk.

"ls –l shows that all my files have zero size.
Have you lost my files?"

TACC

What can go wrong? Unfortunately a lot!

- System error
  - Pretty regularly the tape system thinks that a tape is corrupted
  - Tapes are flagged as damaged
  - This requires the sys-admin to remove the flag

The sls command shows that my files have been stored on a corrupted tape.
Have you lost my files?

TACC

What can go wrong? Unfortunately a lot!

- Corrupted tapes
  - This happens very rarely
  - Tapes are sent to the company for recovery
  - Usually not very much data is recovered

What can go wrong? Unfortunately a lot!

- System abuse by users
  - The disk cache is, of course, much smaller than the storage capacity on tape
  - Users can stage so many files that the cache fills up
  - As a consequence files that are staged to spinning disk are immediately removed from the disk cache to make room for the next file

When I stage files they are so quickly moved back to tape that there is no time to transfer them.
How can I retrieve my files?

Interestingly enough the user that has staged too many files is usually the one that sends the complaint.

Best Practises

- I personally prefer tar files over whole directory trees (compare section on file transfer)
  - Tar files on source (desktop, cluster, etc.)
  - Transfer tar file
- How big may a tar file be?
  - Rule of thumb: Do not make it lager than say 50% of the capacity of a single tape
  - On Ranch the tape capacity is currently 5TB
- A single file can be larger, but it is more difficult to schedule the transfer from/to tape
  - Staging a 10TB file requires access to 2 tape drives simultaneously

Best Practises

- Remove your files when you don’t need them anymore
- Tapes are not that cheap

- We (XSEDE) have now a policy in place. You’ll have to justify your storage needs in your proposal.
- The available spcace is not “unlimited” anymore

TACC

Best Practises

- Why do we have to stage our files?
- Example: 3 files (**`a, b, c`**)
  - Files **`a`** and **`c`** are stored on tape 1
  - File **`b`** is stored on tape 2
- Assume this order of commands:
  - **`scp $ranch:~/a $stampede:~`**
  - **`scp $ranch:~/b $stampede:~`**
  - **`scp $ranch:~/c $stampede:~`**
- Order of operations:
  - Tape 1 is loaded, file a is transferred
  - Tape 2 is loaded, file b is transferred
  - Tape 1 is loaded again, and file c in transferred

Imagine the overhead if you are retrieving thousands of files

TACC

Best Practises

- Solution: stage the files with a special command
- **`stage a b c`**
- The stage command can sort the files
- First files **`a`** and **`c`** are retrieved from tape 1
- Then file **`b`** is retrieved from tape 2
- There is no need to compress the file(s).
  - Automatic hardware compression does that for you
  - Don't worry if your data is already compressed, though

Why using tar files

- Using a few tar files rather than a large number of files in a directory tree allows me to check for file integrity through checksums
- On the source (desktop/cluster) is create the checksum
  - **`md5sum my.tar`**
- On the archive (Ranch) I create also a checksum
  - **`md5sum my.tar > my.tar.md5sum`**
- Before I retrieve the file, I calculated again the checksum and compare against the stored value in file **`my.tar.md5sum`**

Part III: Parallel I/O

- Why parallel file systems?
  - Parallel file systems are faster
  - The speed is proportional to the size
- Basic design
  - Good for fast I/O to fewer and larger files
  - Parallel I/O allows multiple MPI tasks to write to the same file(s)
  - Not good for many small files
  - If every MPI task (thousands) is opening and closing many small files → millions of files or created/destroyed
  - Metadata server is serial

If you open and close "millions" of small files, the file system will crash and the sys-admin will cut you off

Parallel I/O: General Approach

1. Libraries: HDF5, NetCDF, SionLib (Juelich) etc.
   – Easy, convenient, powerful, efficient
2. Manually inserting MPI I/O calls in the code

- There is certainly room for hand-written MPI I/O code, and good work in this area has been done in the past and will be done in the future
- However, I'm not so sure if I would recommend writing MPI I/O over using libraries

(This coming from a guy who likes to beat libraries with hand-written code)

Data Formats and Databases

Slides kindly provided by Linda Woodard and Susan Mehringer @ Cornell

Data Preservation and Discovery

- NSF requires a data management plan with all grant proposals
  - Metadata
  - Formats used
  - Data location
  - Discovery and access plans
  - https://confluence.cornell.edu/display/rdmsgweb/Home

- Large Research Projects
  - Personnel
  - Long time horizons
  - Distant collaborators

- Scientific data formats address some of these issues…

Hierarchical Scientific Data Formats

Data Format	Academic Discipline	Parallel I/O	Software Interfaces	Comments
HDF5	2D and higher dimensional data	yes	C, C++, Fortran, Java, Python, Perl, IDL, Matlab, Mathematica	developed at NCSA
NetCDF	Earth Sciences	yes	C, C++, Fortran, Java, Python, Perl, Ruby, IDL, R, Matlab, ArcGIS	developed at UCAR
FITS	Astrophysics	no	C, C++, Fortran, Java, Python, Perl, IDL, R, Matlab, Mathematica	developed at NASA
Silo	General Visualization	yes	VisIt	developed at LLNL

Scientific Data Formats: HDF5

- Versatile data model that can represent complex data objects and metadata
- Portable file format with no limit on the number or size of data objects
- Open software library that runs on platforms from laptops to massively parallel systems
- Integrated performance features that optimize access time and storage space
- Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection

Source: www.hdfgroup.org/hdf5

HDF5 Features

- *Headers* include extensive metadata (datatypes, dimensionality, storage layout); files are self-documenting
- *Virtual file layer* provides flexible storage and transfer capabilities: Standard (Posix), Parallel, and Network I/O file drivers
- *Compression & chunking* increase access and storage efficiency
- *Datatype transformations* can be performed during I/O operations
- *Subsetting* reduces transferred data volume & improves access speed during I/O operations

Source: www.hdfgroup.org/hdf5

Scientific Data Formats: netCDF

- Similar to HDF5; newest version uses the HDF5 format
- Used extensively in the Earth Sciences community for time varying geospatial data; most data from NOAA is in netCDF format
- NetCDF has good tools for geo-gridded data
  - *Panoply* (http://www.giss.nasa.gov/tools/panoply/) focuses on the presentation of geo-gridded data.
  - *Ferret* (http://ferret.wrc.noaa.gov/Ferret/) offers a Mathematica-like approach to analysis. Variables and expressions may be defined interactively; calculations may be applied over arbitrarily shaped regions; geophysical formatting is built in.
  - *Parallel-NetCDF* (http://trac.mcs.anl.gov/projects/parallel-netcdf/) is built upon MPI-IO to distribute file reads and writes efficiently among multiple processors.

netCDF Features

- *Self-Describing*—files include information about the data they contain
- *Portable*—endian problems handled automatically
- *Direct-access*—subsets of a larger dataset can be accessed without reading through all the preceding data
- *Appendable*—data may be appended to a properly structured netCDF file without copying the dataset or redefining its structure
- *Shareable*—one writer and multiple readers may simultaneously access the same netCDF file
- *Archivable*—netCDF will always be backwards compatible

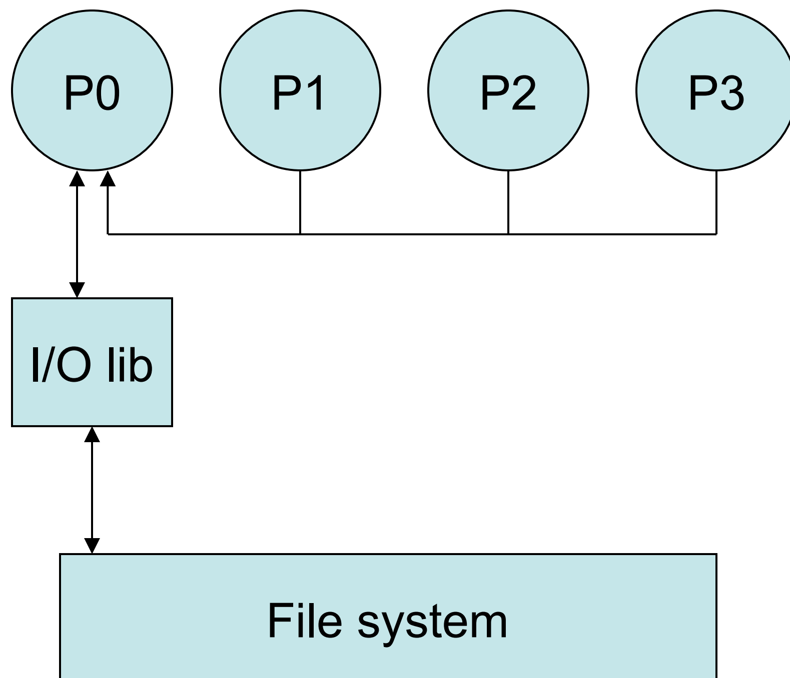
Source: http://www.unidata.ucar.edu/software/netcdf

Scientific Data Formats: Silo

- Silo is a library for reading and writing scientific data to binary disk files
- Silo supports point meshes, structured and unstructured meshes in 2D and 3D
- Two layers
  - API with Fortran, C, and Python interfaces
  - I/O driver (HDF5 is one of these drivers)
- Primary file format for VisIt

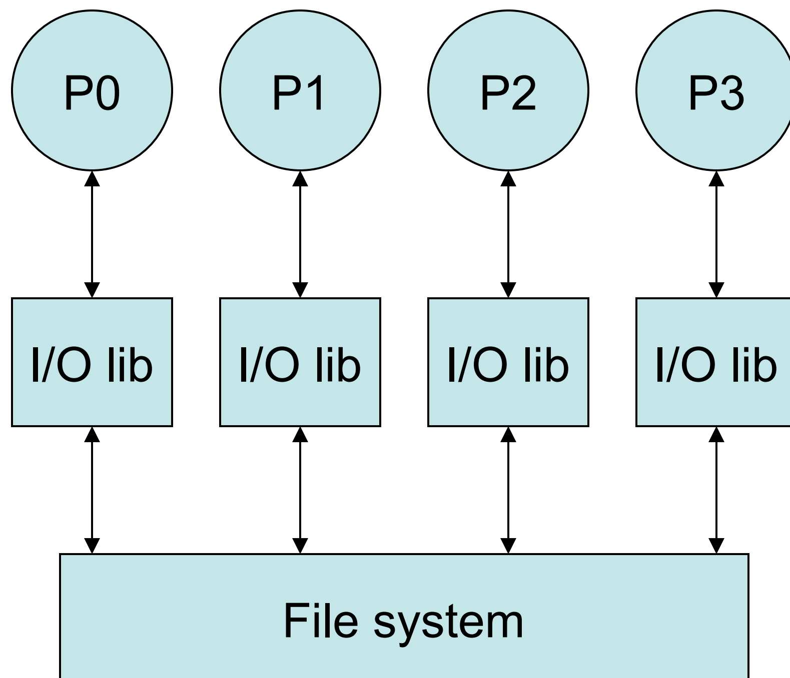
https://wci.llnl.gov/codes/silo/index.html

Why Parallel I/O is important



- P0 may become bottleneck
- System memory may be exceeded on P0

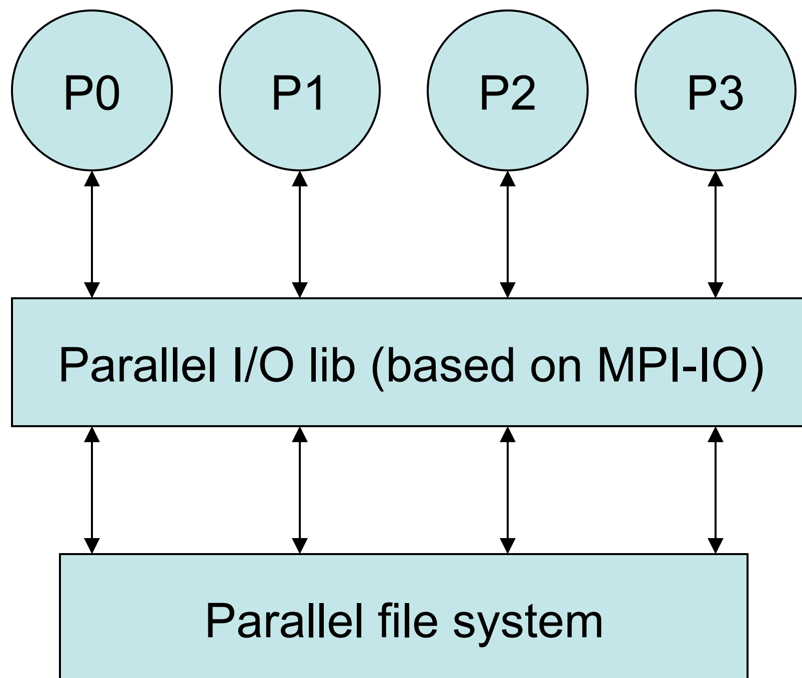
Why Parallel I/O is important – part 2



- Possible to achieve good performance
- May require post-processing
- More work for applications, programmers

Cornell University
Center for Advanced Computing

Why Parallel I/O is important – part 3

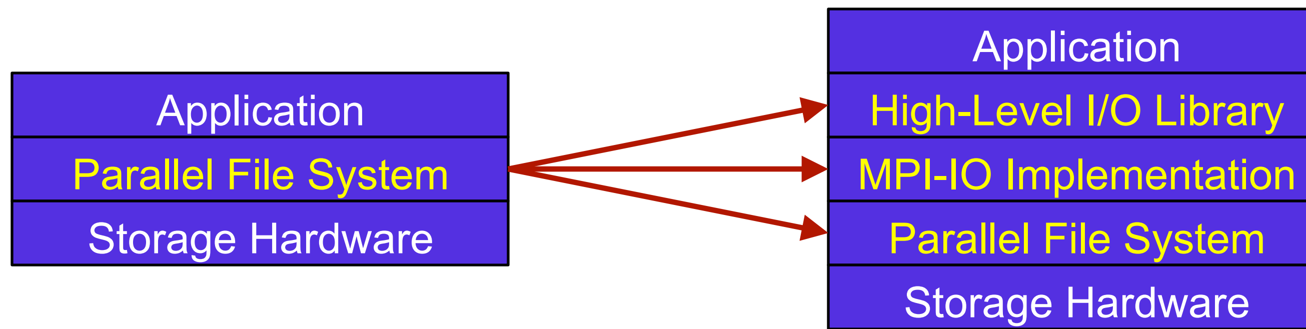


- HDF5, netCDF and Silo can take the place of a parallel I/O library - they've linked the parallel I/O library for you
- Variant: only P1 and P2 act as parallel writers; they gather data from P0 and P3 respectively (chunking)

Lustre

Slides kindly provided by Steve Lantz and Susan Mehringer @ Cornell

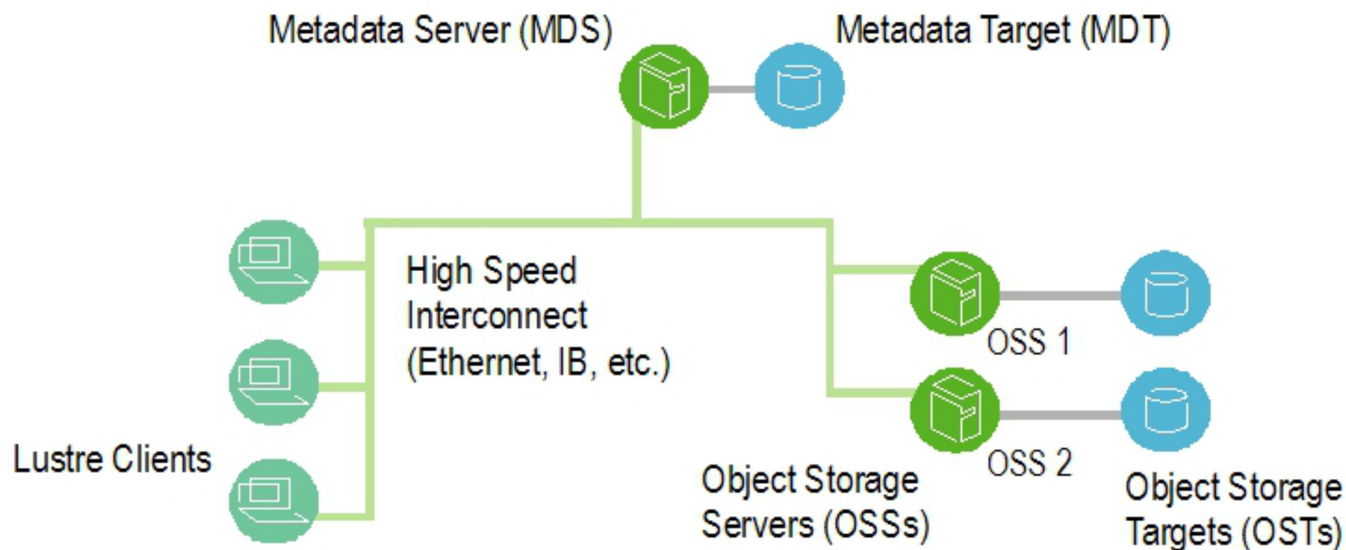
Introduction: The Parallel I/O Stack



- Parallel I/O can be hard to coordinate and optimize
- Solution is to have specialists implement several intermediate layers
  - High-level I/O library maps application abstractions to a structured, portable file format (e.g., HDF5, Parallel netCDF)
  - Middleware layer deals with organizing access by many processes (e.g., MPI-IO, UPC-IO)
  - Parallel file system maintains logical file space, provides efficient access to data (e.g., Lustre, PVFS, GPFS)

Lustre Components

- All large TACC file systems are Lustre, which is a globally available distributed file system.
- Primary components are the MDS and OSS nodes. The OSSs contain the data, while the MDS contains the filename-to-object map.

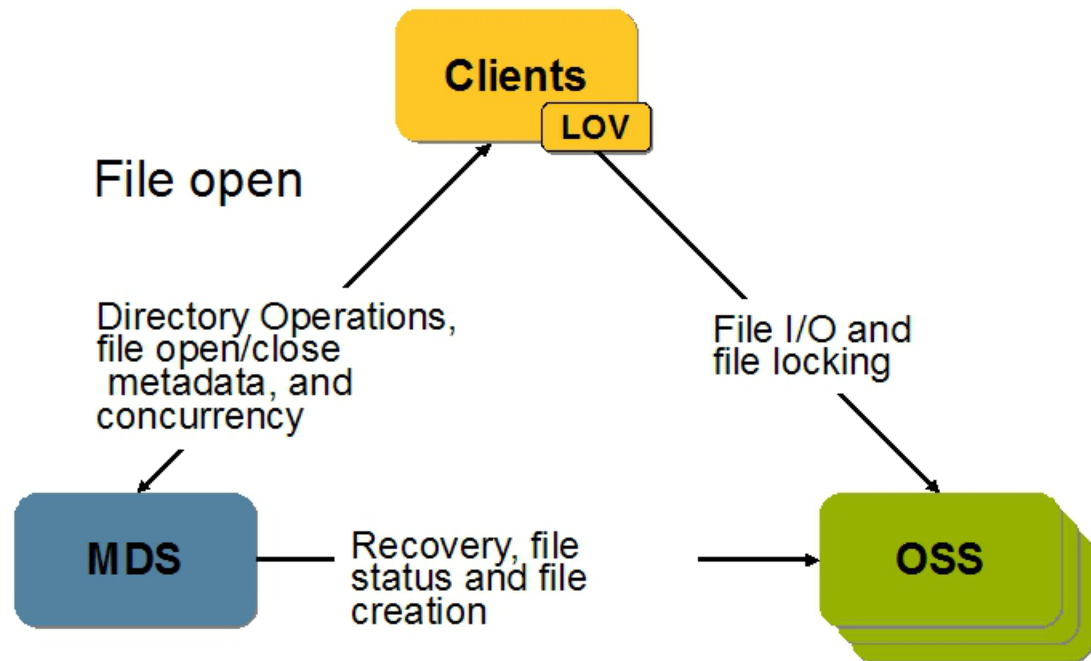


http://wiki.lustre.org/manual/LustreManual18_HTML/IntroductionToLustre.html#50651242_pgfId-1287192

Cornell University
Center for Advanced Computing

Parts of the Lustre System

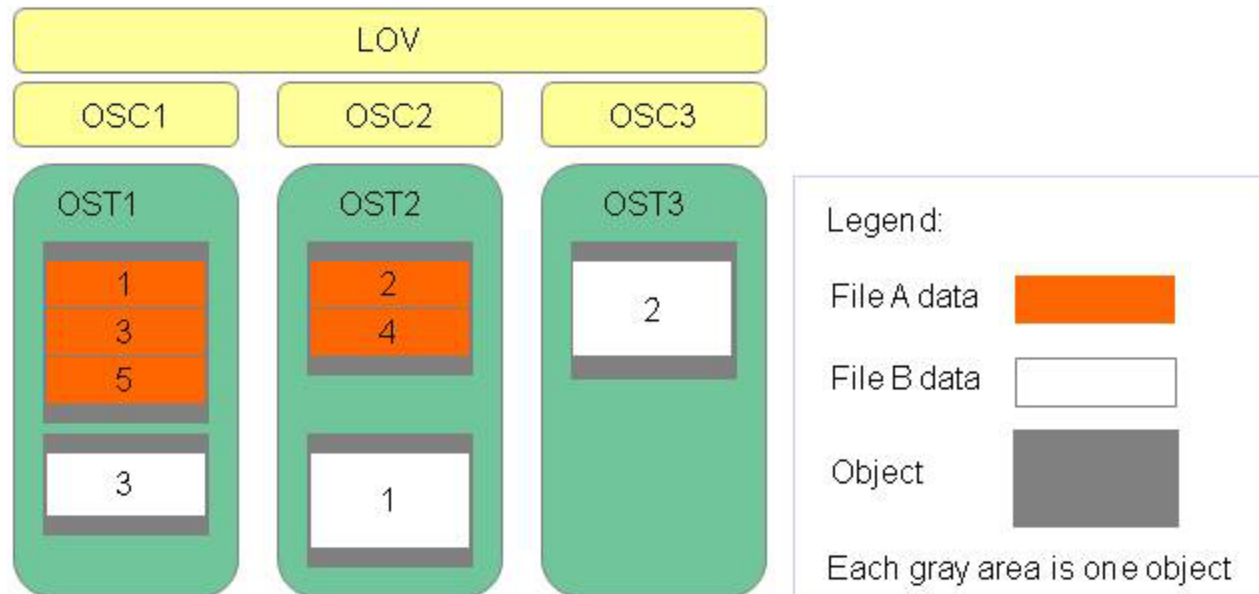
- The client (you) must talk to both the MDS and OSS servers in order to use the Lustre system.
- File I/O goes to one or more OSS's. Opening files, listing directories, etc. go to the MDS.
- Front end to the Lustre file system is a Logical Object Volume (LOV) that simply appears like any other large volume that would be mounted on a node.



Cornell University
Center for Advanced Computing

Lustre File System and Striping

- Striping allows parts of files to be stored on different OSTs, in a RAID-0 pattern.
  - The number of objects is called the stripe_count.
  - Objects contain "chunks" of data that can be as large as stripe_size.



Benefits of Lustre Striping

- Due to striping, the Lustre file system scales with the number of OSS's available.
- The capacity of a Lustre file system equals the *sum* of the capacities of the storage targets.
  - Benefit #1: max file size is not limited by the size of a single target.
  - Benefit #2: I/O rate to a file is the of the aggregate I/O rate to the objects.
- Ranger provides 72 Sun I/O nodes, with an nominal data rate that approaches 50GB/s, but this speed is split by all users of the system.
- Metadata access can be a bottleneck, so the MDS needs to have especially good performance (e.g., solid state disks on some systems).

Lustre File System (lfs) Commands

- Among various lfs commands are lfs getstripe and lfs setstripe.
- The lfs setstripe command takes four arguments:

```
lfs setstripe
 <file|dir> -s <bytes/OST> -o <start OST> -c <#OSTs>
```

1. File or directory for which to set the stripe.
2. The number of bytes on each OST, with k, m, or g for KB, MB or GB.
3. OST index of first stripe (-1 for filesystem default) .
4. Number of OSTs to stripe over.

- So to stripe across two OSTs, you would call:

```
lfs setstripe bigfile -s 4m -o -1 -c 2
```

Getting Properties of File Systems and Files

- There are lfs commands to tell you the quotas and striping for Lustre file systems and files. Get the quota for $WORK with

```
lfs quota $WORK
```

- To see striping, try creating a small file and then using lfs to get its stripe information.

```
ls > file.txt
lfs getstripe file.txt
```

- The listing at the end of the results shows which OSTs have parts of the file.

A Striping Test to Try

- You can set striping on a file or directory with the lfs setstripe command. First set it for a file:

```
lfs setstripe stripy.txt -s 4M -o -1 -c 6
ls -la > stripy.txt
lfs getstripe stripy.txt
```

- Now try the same thing for a directory. First create a directory, then set its striping, then make a file within that directory.

```
mkdir s; cd s; lfs setstripe . -s 4M -o -1 -c 6
ls -la > file.txt
lfs getstripe file.txt
```

- In both cases, you should see the file striped across six OSTs.

Parallel I/O (MPI-2)

Slides kindly provided by Steve Lantz and Susan Mehringer @ Cornell

Parallel I/O with MPI-IO

- Why parallel I/O?
  - I/O was lacking from the MPI-1 specification
  - Due to need, it was defined independently, then subsumed into MPI-2
- What is parallel I/O? It occurs when:
  - multiple MPI tasks can read or write simultaneously,
  - from or to a single file,
  - in a parallel file system,
  - through the MPI-IO interface.
- A parallel file system works by:
  - appearing as a normal Unix file system, while
  - employing multiple I/O servers (usually) for high sustained throughput.

MPI-IO Advantages

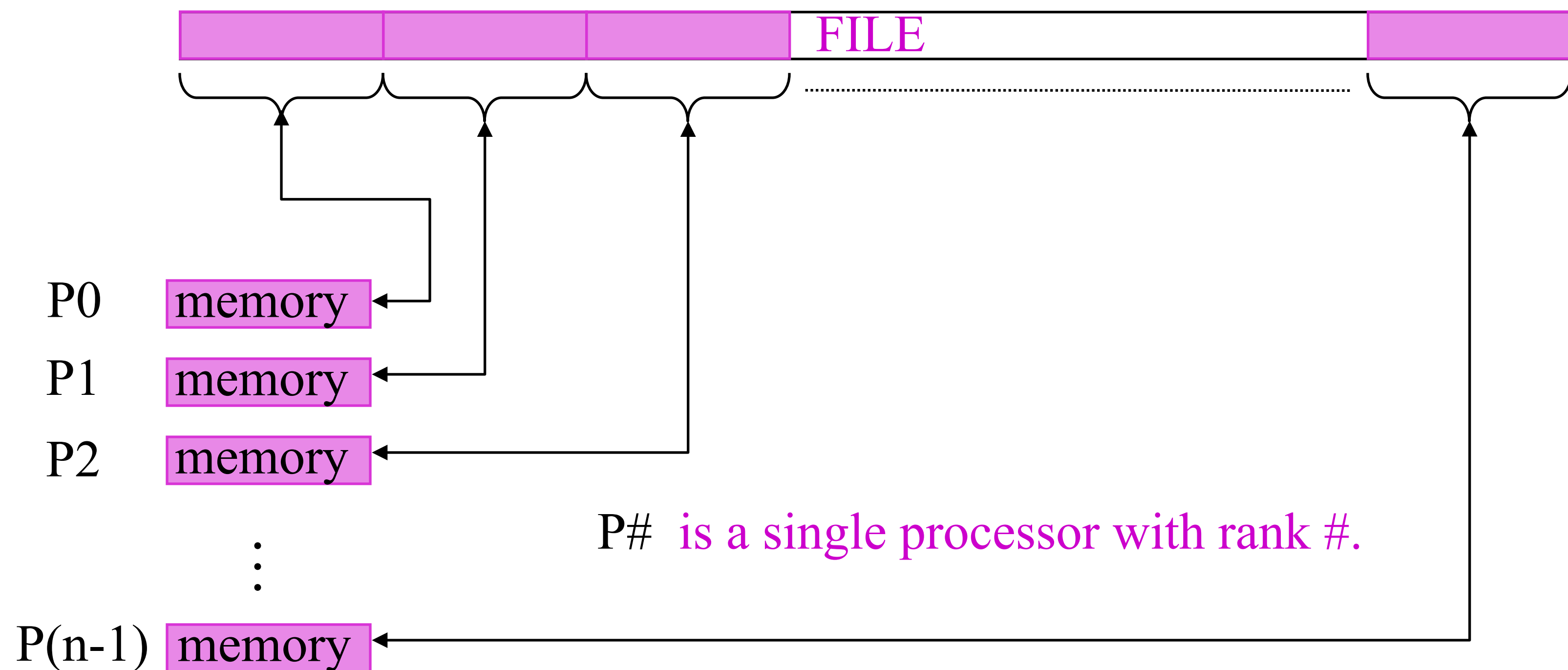
- Two common alternatives to parallel MPI-IO are:
  1. Rank 0 accesses a file; it gathers/scatters file data from/to other ranks.
  2. Each rank opens a separate file and does I/O to it independently.
- Alternative I/O schemes are simple enough to code, but have either
  1. Poor scalability (e.g., the single task is a bottleneck) or
  2. File management challenges (e.g., files must be collected from local disk).
- MPI-IO provides
  – mechanisms for performing synchronization,
  – syntax for data movement, and
  – means for defining noncontiguous data layout in a file (MPI datatypes).

Noncontiguous Accesses

- Parallel applications commonly need to write distributed arrays to disk
  - Better to do this to a single file, instead of multiple
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in both a file **and** a memory buffer.
  - Read or write such a file in parallel by using derived datatypes within a single MPI function call
  - Let the MPI implementation optimize the access
- Collective I/O combined with noncontiguous accesses generally yields the highest performance
- HPC parallel I/O requires some extra work, but it
  - potentially provides high throughput and
  - offers a single (unified) file for viz and pre/post processing

Simple MPI-IO

Each MPI task reads/writes a single block:



File Pointers and Offsets

- In simple MPI-IO, each MPI process reads or writes a single block.
- I/O functions must be preceded by a call to MPI_File_open, which defines both an *individual* file pointer for the process, and a *shared* file pointer for the communicator.
- We have three means of positioning where the read or write takes place for each process:
  1. Use individual file pointers, call MPI_File_seek/read
  2. Calculate byte offsets, call MPI_File_read_at
  3. Access a shared file pointer, call MPI_File_seek/read_shared
- Techniques 1 and 2 are naturally associated with C and Fortran, respectively. In any case, the goal is roughly indicated by the previous figure.

Reading by Using Individual File Pointers – C Code

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints   = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              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);
MPI_File_close(&fh);
```

Reading by Using Explicit Offsets – F90 Code

```
include 'mpif.h'
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset

nints  = FILESIZE/(nprocs*INTSIZE)
offset = rank * nints * INTSIZE

call MPI_FILE_OPEN( MPI_COMM_WORLD, '/pfs/datafile', &
                    MPI_MODE_RDONLY,                 &
                    MPI_INFO_NULL, fh, ierr)
call MPI_FILE_READ_AT( fh, offset, buf, nints,
                       MPI_INTEGER, status, ierr)
call MPI_FILE_CLOSE(fh, ierr)
```

Operations with Pointers, Offsets, Shared Pointers

- MPI_File_open flags:
  - **`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)
  - Use bitwise-or '|' in C, or addition '+" in Fortran, to combine multiple flags
- To write into a file, use MPI_File_write or MPI_File_write_at, or…
- The following operations reference the implicitly-maintained shared pointer defined by MPI_File_open
  - **`MPI_File_read_shared`**
  - **`MPI_File_write_shared`**
  - **`MPI_File_seek_shared`**

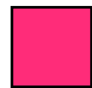
File Views

- A *view* is a triplet of arguments (*displacement*, *etype*, *filetype*) that is passed to **MPI_File_set_view.**

  - *displacement* = number of bytes to be skipped from the start of the file
  - *etype* = unit of data access (can be any basic or derived datatype)
  - *filetype* = specifies layout of etypes within file

- Note that etype is considered to be the elementary type, but since it can be a derived datatype, there’s really nothing elementary about it.
- In the file view depicted on the next slide, etype is double precision, filetype is a vector type, and displacement is used to stagger the starting positions by MPI rank.

Example #1: File Views for a Four-Task Job

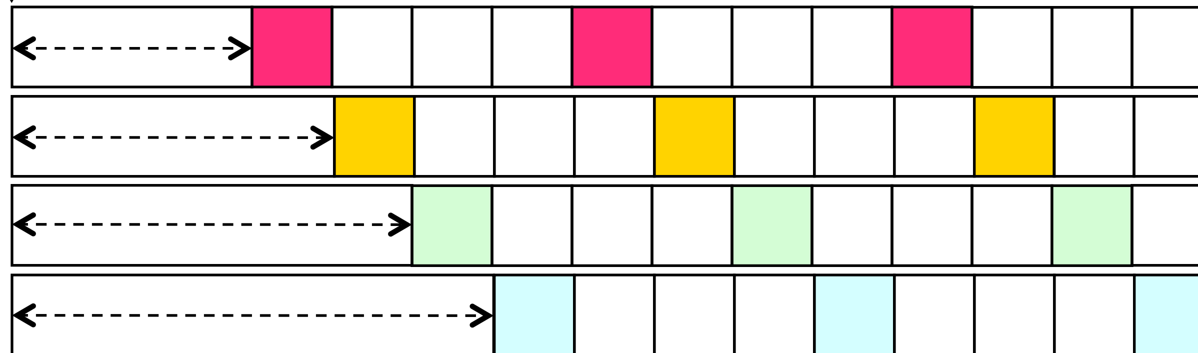


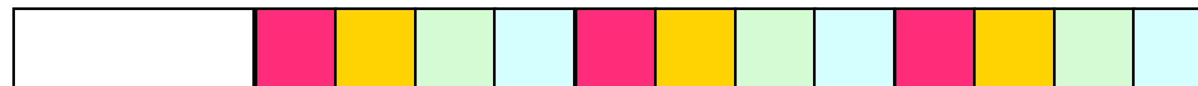
`etype = MPI_DOUBLE_PRECISION` elementary datatype

`filetype = myPattern` derived datatype, sees every 4th DP

head of file

VIEW: each task repeats myPattern with different displacements





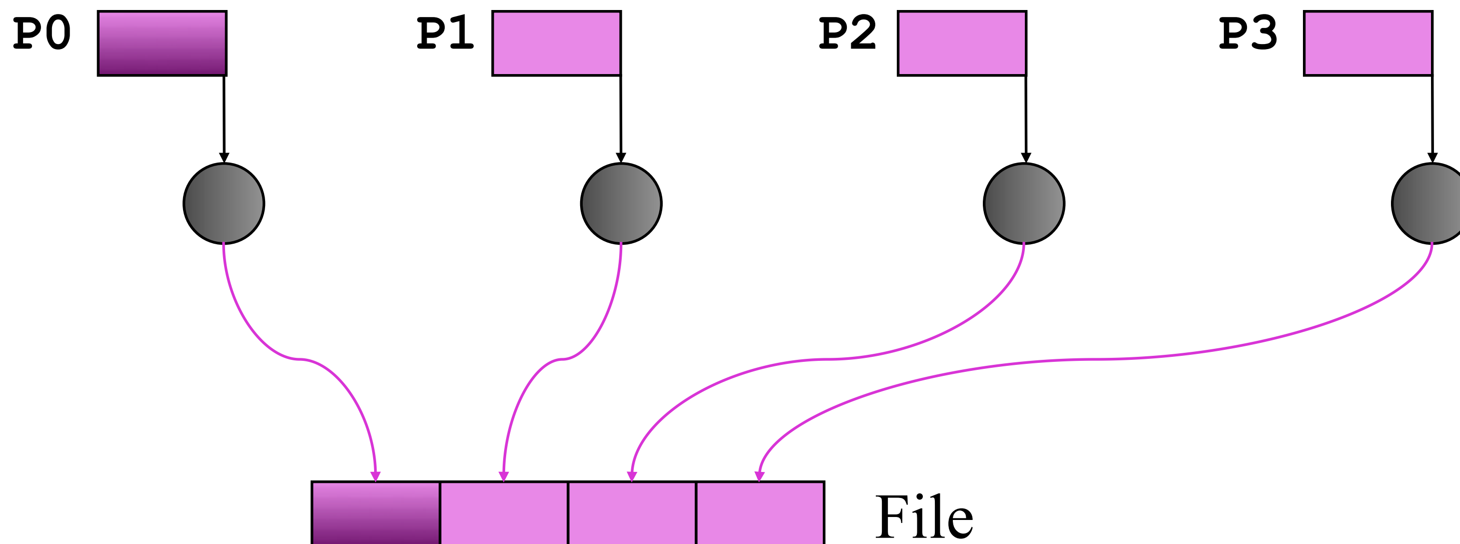
File View Examples

- In Example 1, we write contiguous data into a contiguous block defined by a file view.
  - We give each process a different file view so that together, the processes lay out a series of blocks in the file, one block per process.

- In Example 2, we write contiguous data into two *separate* blocks defined by a different file view.
  - Each block is a contiguous type in memory, but the pair of blocks is a *vector* type in the file view.
  - We again use displacements to lay out a series of blocks in the file, one block per process, in a repeating fashion.

Example #1: File Views for a Four-Task Job

- 1 block from each task, written in task order



`MPI_File_set_view` assigns regions of the file to separate processes

Code for Example #1

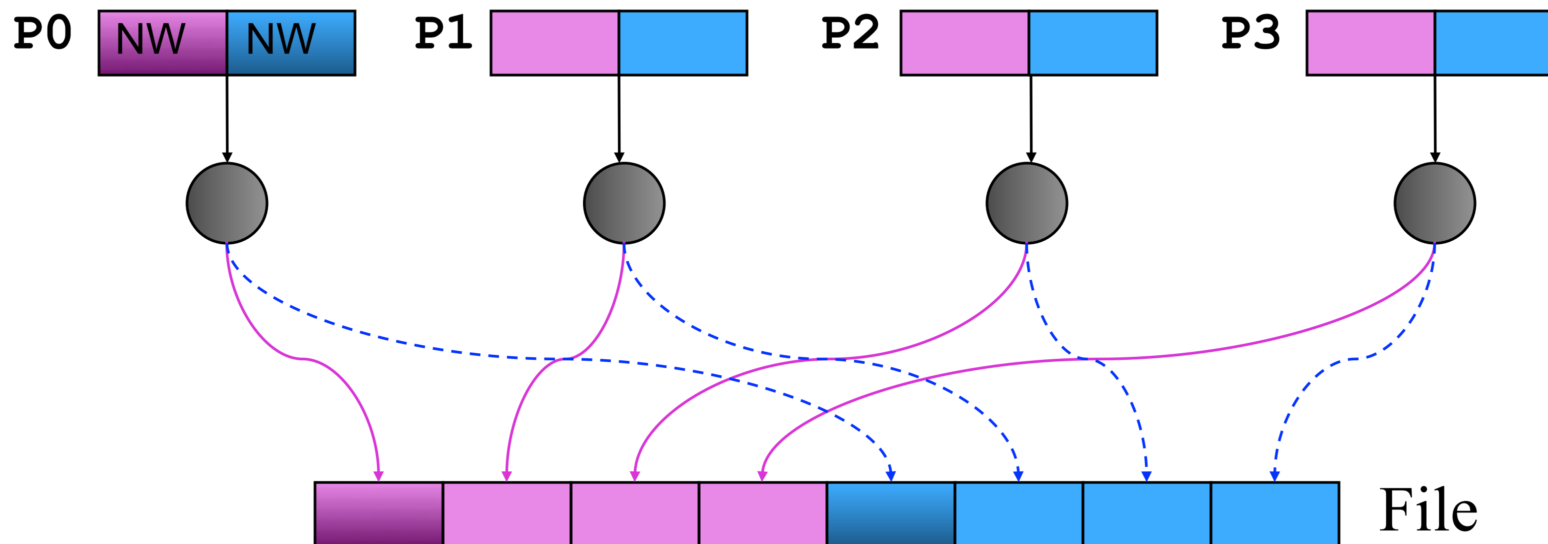
```
#define N 100
MPI_Datatype arraytype;
MPI_Offset disp;

disp = rank*sizeof(int)*N; etype = MPI_INT;
MPI_Type_contiguous(N, MPI_INT, &arraytype);
MPI_Type_commit(&arraytype);

MPI_File_open(    MPI_COMM_WORLD, "/pfs/datafile",
                  MPI_MODE_CREATE | MPI_MODE_RDWR,
                  MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, arraytype,
                  "native", MPI_INFO_NULL);
MPI_File_write(fh, buf, N, etype, MPI_STATUS_IGNORE);
```

Example #2: File Views for a Four-Task Job

- 2 blocks from each task, written in round-robin fashion to a file



`MPI_File_set_view` assigns regions of the file to separate processes

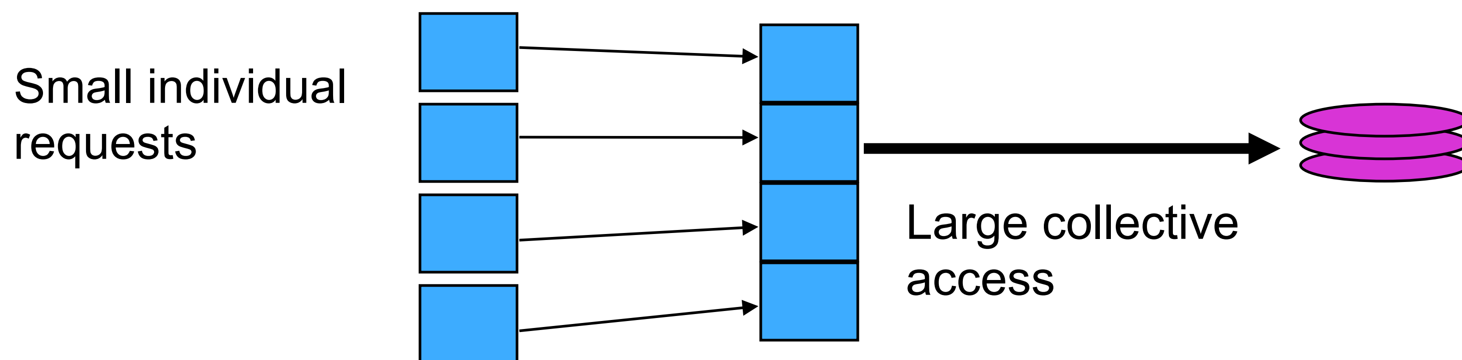
Code for Example #2

```
int buf[NW*2];
   MPI_File_open(MPI_COMM_WORLD, "/data2",
                 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
/* want to see 2 blocks of NW ints, NW*npes apart */
   MPI_Type_vector(2, NW, NW*npes, MPI_INT, &fileblk);
   MPI_Type_commit(                         &fileblk);
   disp = (MPI_Offset)rank*NW*sizeof(int);
   MPI_File_set_view(fh, disp, MPI_INT, fileblk,
                     "native", MPI_INFO_NULL);

/* processor writes 2 'ablk', each with NW ints */
   MPI_Type_contiguous(NW,   MPI_INT, &ablk);
   MPI_Type_commit(&ablk);
   MPI_File_write(fh, (void *)buf, 2, ablk, &status);
```

Collective I/O in MPI

- A critical optimization in parallel I/O
- Allows communication of “big picture” to file system
- Framework for 2-phase I/O, in which communication precedes I/O
- Preliminary communication can use MPI machinery to aggregate data
- Basic idea: build large blocks, so that reads/writes in I/O system will be more efficient



MPI Routines for Collective I/O

- Typical routine names:
  - **`MPI_File_read_all`**
  - **`MPI_File_read_at_all`**, etc.
- The _all indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
- Each process provides nothing beyond its own access information, including its individual pointer
  - The argument list is therefore the same as for the non-collective functions
- Collective I/O operations work with shared pointers, too
  - The general rule is to replace _shared with _ordered in the routine name
  - Thus, the collective equivalent of MPI_File_read_shared is MPI_File_read_ordered

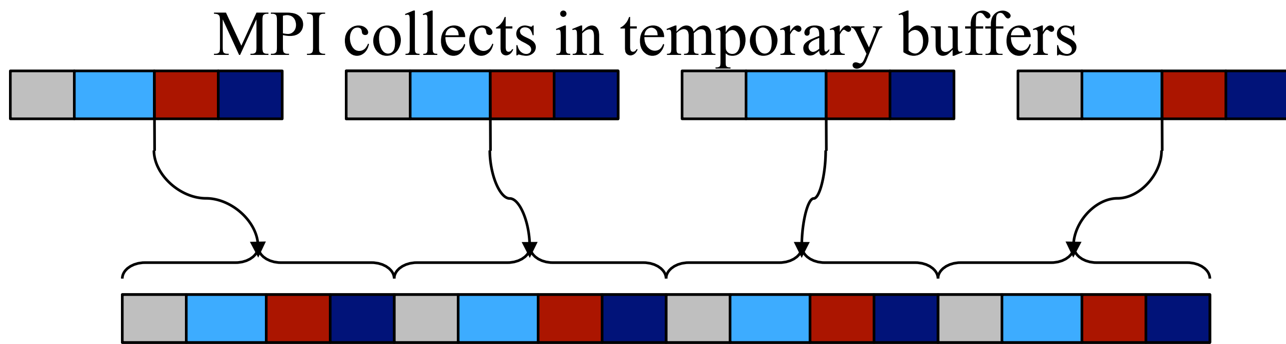
Advantages of Collective I/O

- By calling the collective I/O functions, the user allows an implementation to optimize the request based on the combined requests of all processes
- The implementation can merge the requests of different processes and service the merged request efficiently
- Particularly effective when the accesses of different processes are *noncontiguous* and *interleaved*

Collective Choreography

Original memory layout on 4 processors

MPI collects in temporary buffers



then writes to File layout

Asynchronous Operations

Asynchronous operations give the system even more opportunities to optimize I/O.

For each *noncollective* I/O routine, there is an *nonblocking* variant.

- MPI_File_iwrite and MPI_File_iread, e.g., are nonblocking calls.
- The general naming convention is to replace “read” with “iread”, or “write” with “iwrite”.
- These nonblocking routines are analogous to the nonblocking sends and receives in MPI point-to-point communication.
- Accordingly, these types of calls should be terminated with MPI_Wait.

Collective Asynchronous Operations

For each *collective* I/O routine, there is a *split* variant.

- A collective I/O operation can *begin* at some point and *end* at some later point.
- When using file pointers:
  - **`MPI_File_read_all_begin/end`**
  - **`MPI_File_write_all_begin/end`**
- When using explicit offsets:
  - **`MPI_File_read_at_all_begin/end`**
  - **`MPI_File_write_at_all_begin/end`**
- When using shared pointers:
  - **`MPI_File_read_ordered_begin/end`**
  - **`MPI_File_write_ordered_begin/end`**

Summary of Variants for MPI_File_...

MPI-IO envisions three orthogonal aspects to data access:

- Positioning – explicit offset vs. file pointer (individual or shared)
- Synchronism – blocking vs. nonblocking/split
- Coordination – noncollective vs. collective

The table summarizes all 12 combinations and naming conventions.

	blocking	nonblocking	collective	split collective
Explicit offsets	`read_at`	`iread_at` `wait`	`read_at_all`	`read_at_all_begin` `read_at_all_end`
Individual pointers	`read`	`iread` `wait`	`read_all`	`read_all_begin` `read_at_all_end`
Shared pointer	`read_shared`	`iread_shared` `wait`	`read_ordered`	`read_ordered_begin` `read_ordered_end`

Passing Along Hints to MPI-IO

```
MPI_Info info;
MPI_Info_create(&info);

/* no. of I/O devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR,
              info, &fh);

MPI_Info_free(&info);
```

Examples of Hints (also used in ROMIO)

- `striping_unit`
- `striping_factor`
- `cb_buffer_size`
- `cb_nodes`

MPI-2 predefined hints

- `ind_rd_buffer_size`
- `ind_wr_buffer_size`

New algorithm parameters

- `start_iodevice`
- `pfs_svr_buf`
- `direct_read`
- `direct_write`

Platform-specific hints

MPI-IO Summary

- MPI-IO has many features that can help users achieve high performance
- The most important of these features are:
  - the ability to specify noncontiguous accesses
  - the collective I/O functions
  - the ability to pass hints to the implementation
- In particular, when accesses are noncontiguous, users must:
  - Create derived datatypes
  - Define file views
  - Use the collective I/O functions
- Use of these features is encouraged, because I/O is expensive! It's best to let the system make tuning decisions on your behalf.

MPI-IO Is Not Your Only Choice for an API…

A few higher-level alternatives exist. Note that all are built upon MPI-IO.

- Parallel HDF5
- NetCDF 4 Parallel (requires Parallel HDF5)
- ADIOS, the ADaptable IO System from ORNL, Georgia Tech, Rutgers
  - Was created collaboratively with several major HPC simulation groups
  - Resembles standard Fortran POSIX I/O routines
  - Supports Parallel HDF5 and NetCDF 4 as options
  - Stores metadata about data hierarchies, data types, data organization, process groupings, etc., an auxiliary XML file
  - Lets the user select the I/O to be individual, collective, asynchronous, etc., via the XML file, rather than by recoding and recompiling

Choose to use one (or none) of these based on your application's needs.

Summary

- Be aware that “Big Data” comes with its own set of problems
- Know that you can easily impede other users, or the file system
- Relevant resources are constantly monitored by the sys-admins (even more so than other parts of the HPC infrastructure)
- Consider using libraries for parallel I/O
- Know that unoptimized (non-parallel, etc.) I/O may stop your code from scaling