Parallel I/O

Steve Lantz
Senior Research Associate
Cornell CAC

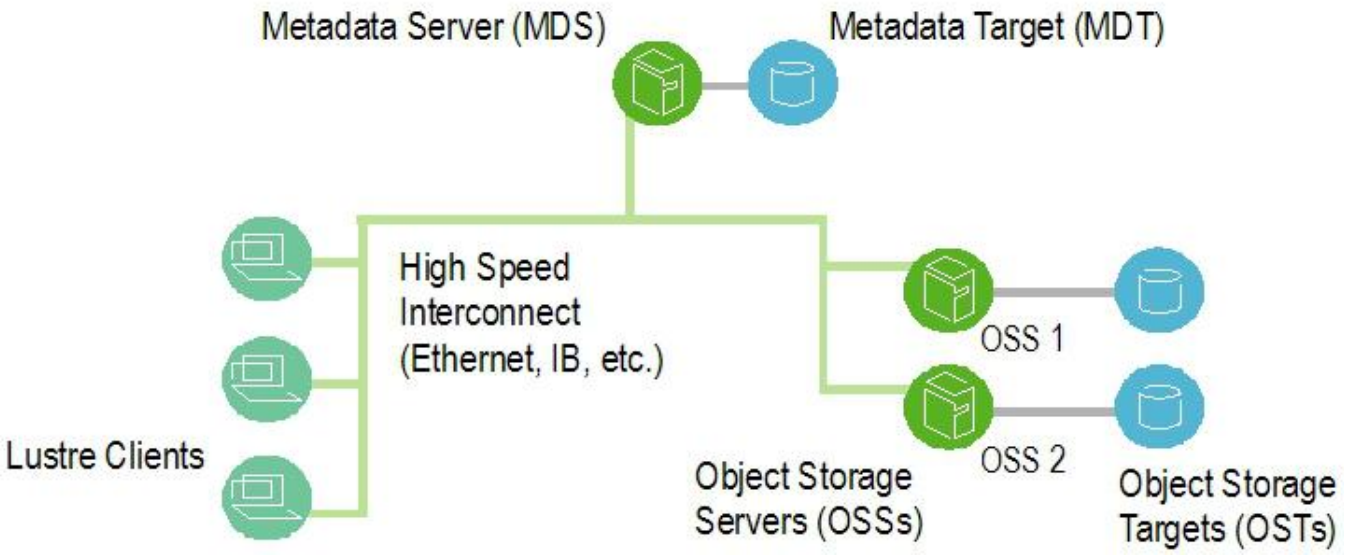
Workshop: Data Analysis on Ranger, January 19, 2012

Based on materials developed by Bill Barth at TACC

1. Lustre

Lustre Components

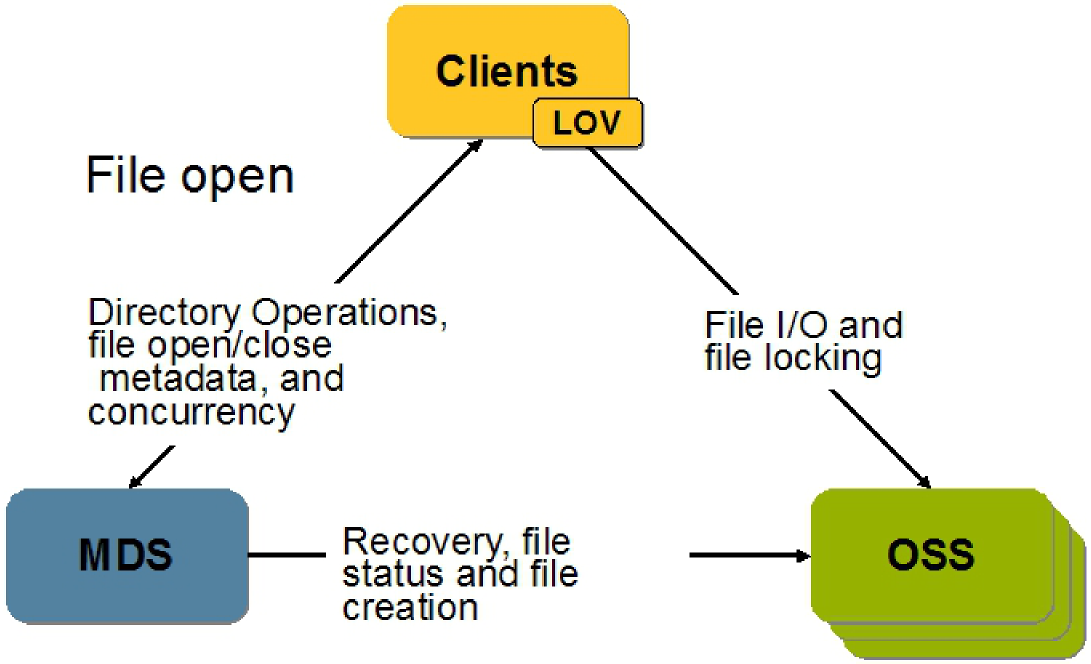
- All Ranger file systems are Lustre, which is a globally available distributed file system.
- Primary components are the MDS and OSS nodes. The OSSs contain the data, while the MDS contains the filename-to-object map.



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

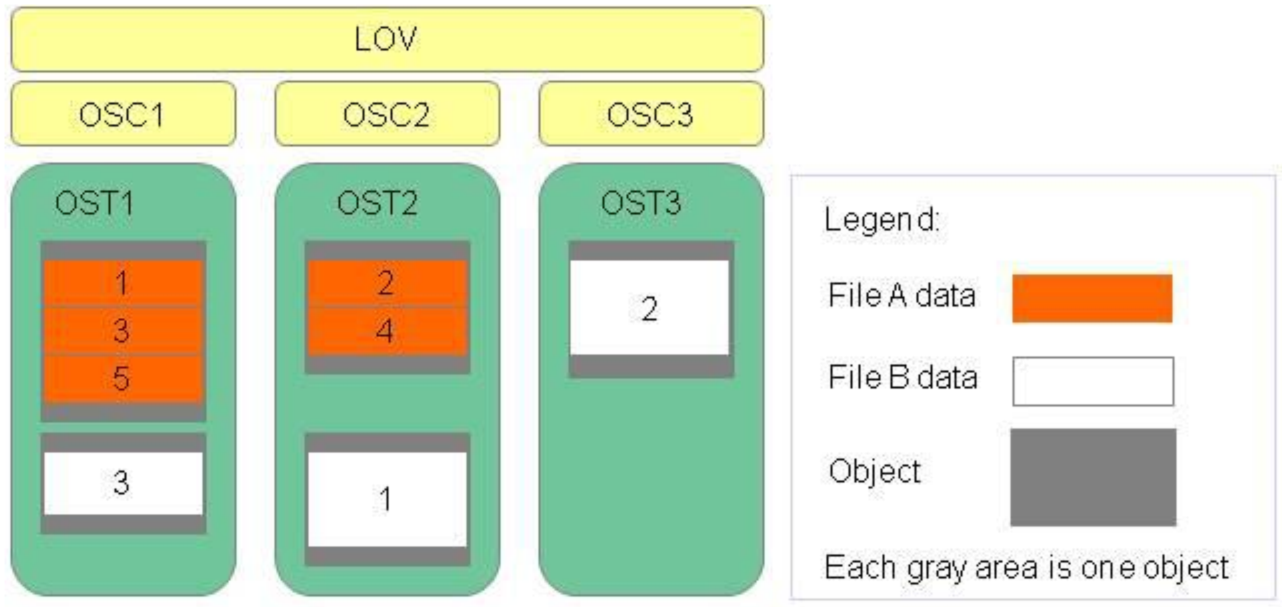
Parts of the Lustre System

- 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.


Clients
LOV
File open
Directory Operations, file open/close metadata, and concurrency
File I/O and file locking
MDS
Recovery, file status and file creation
OSS

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.

2. Parallel I/O (MPI-2)

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

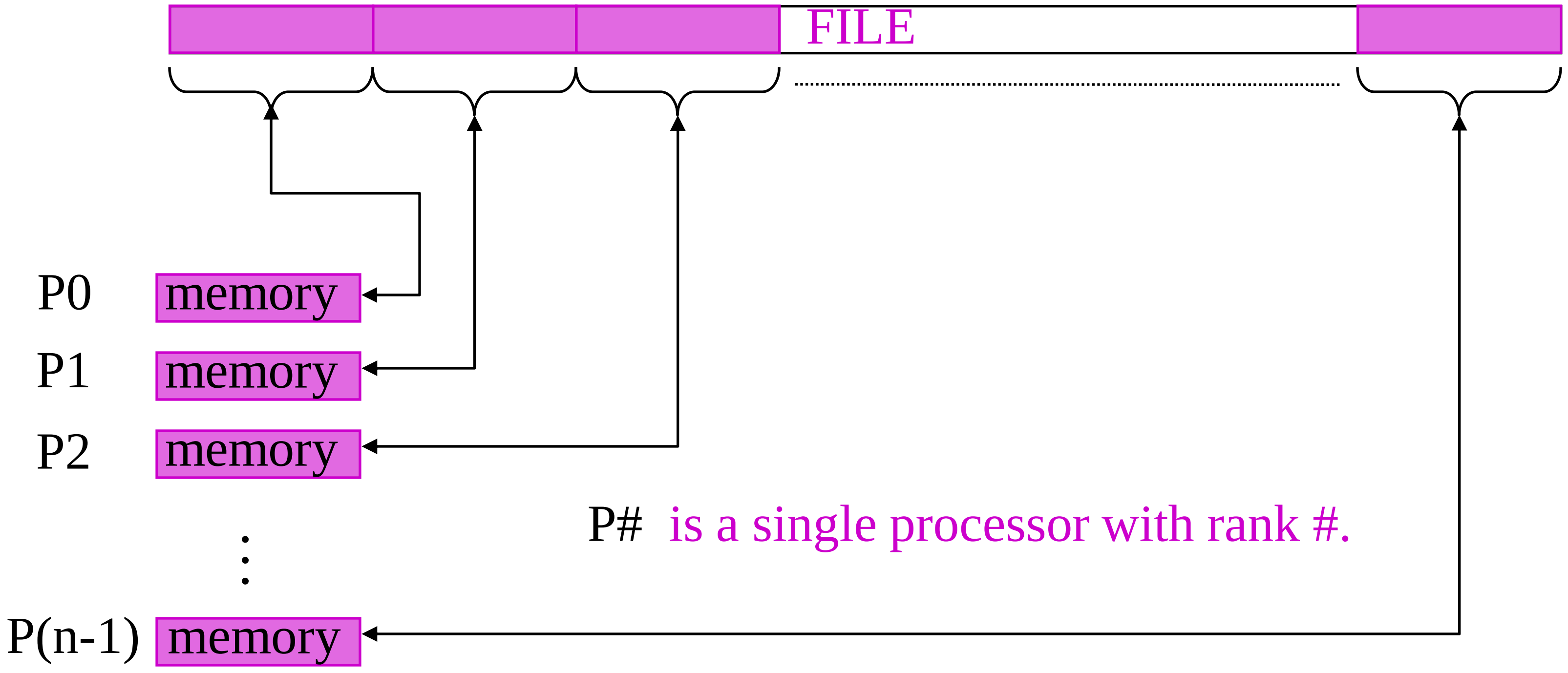
- Two common alternatives to parallel MPI-IO are:
  1. Rank 0 accesses a file; it gathers/scatters file data from/to other ranks.
  2. Each rank opens a separate file and does I/O to it independently.
- Alternative I/O schemes are simple enough to code, but have either
  1. Poor scalability (e.g., the 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 to 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)
```

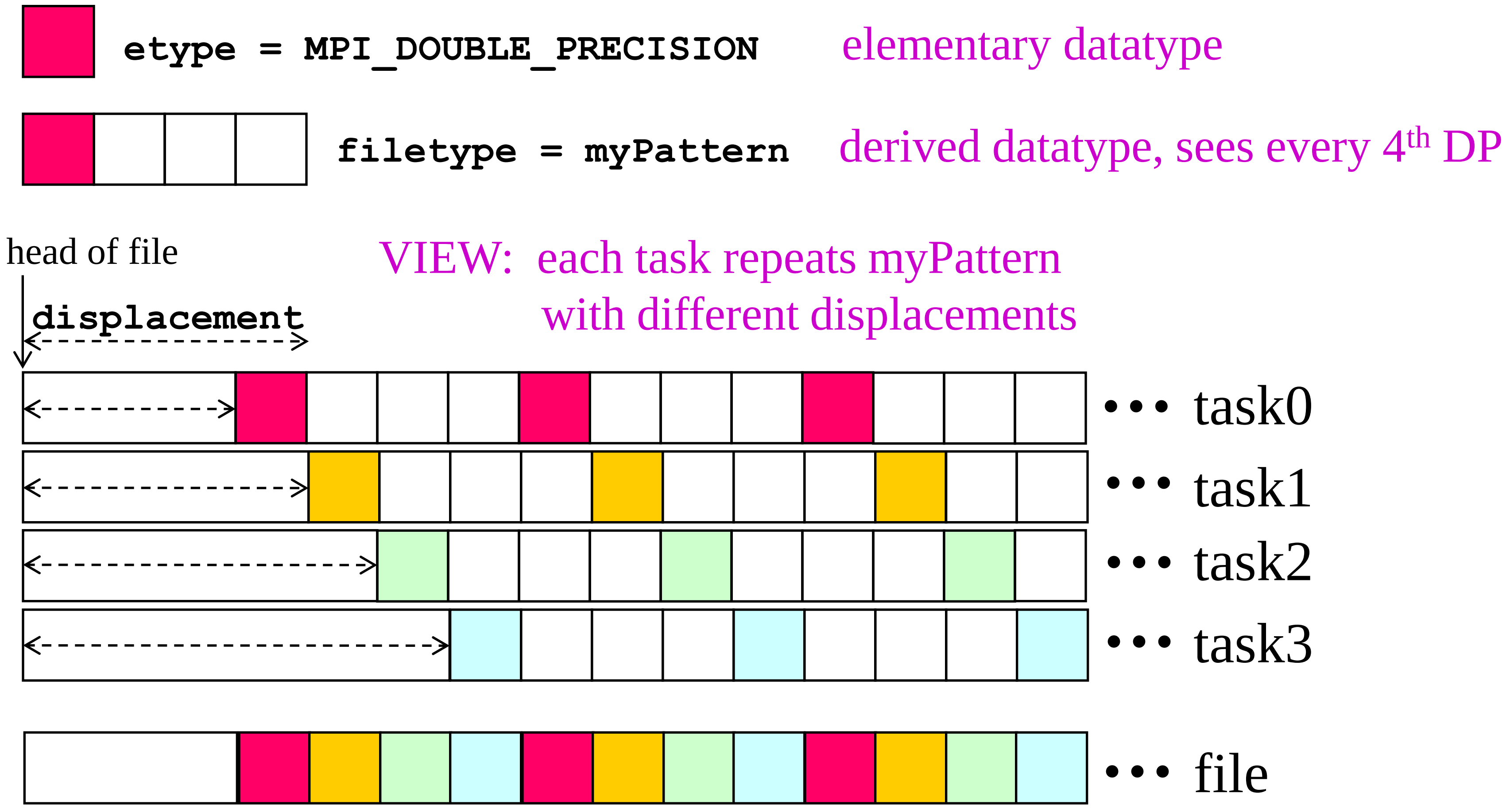
Operations with Pointers, Offsets, Shared Pointers

- 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
displacement
VIEW: each task repeats myPattern with different displacements
••• task0
••• task1
••• task2
••• task3
••• file

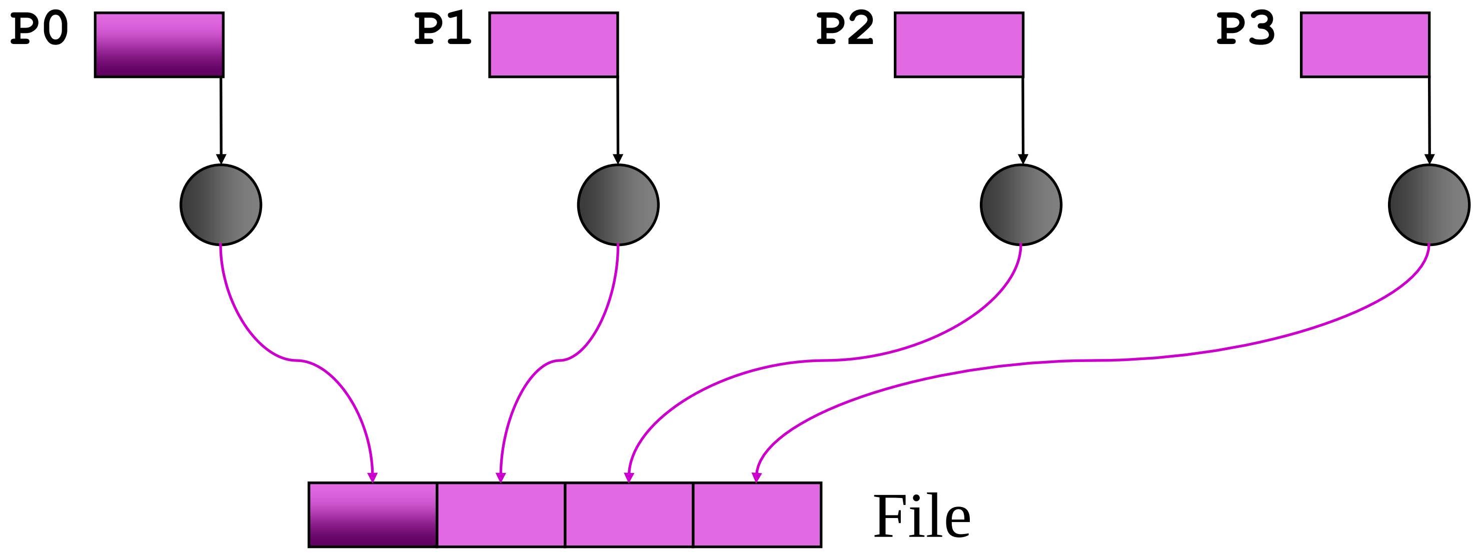
File View Examples

- 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

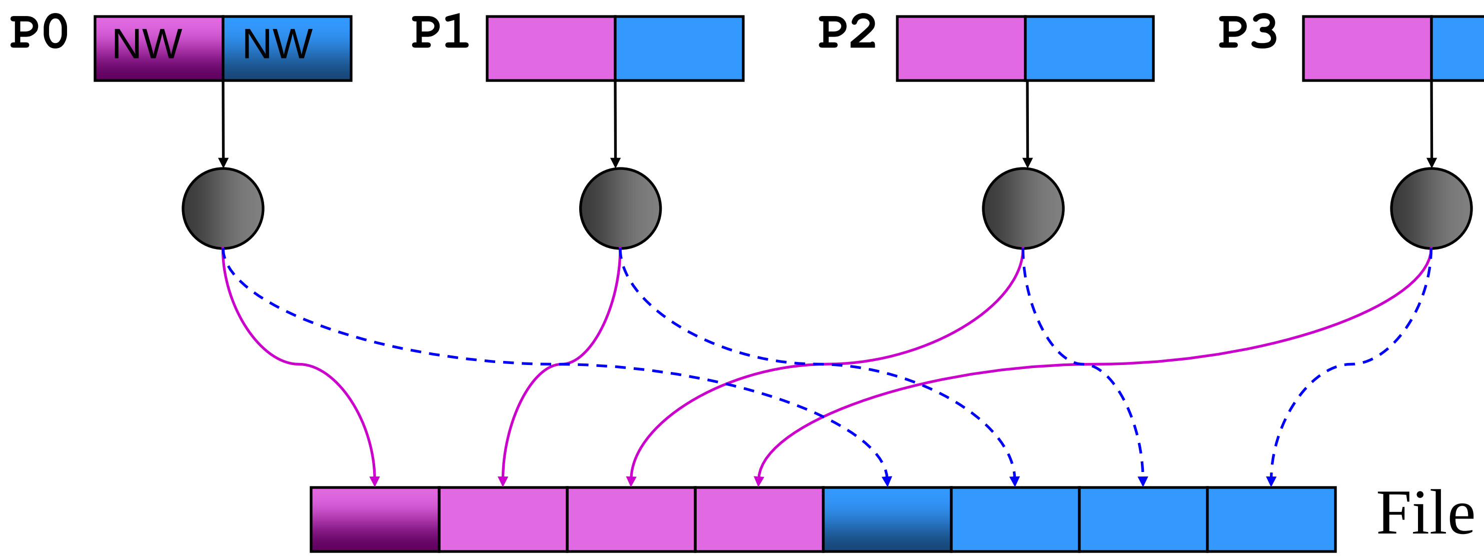
```
#define N 100
MPI_Datatype arraytype;
MPI_Offset disp;

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

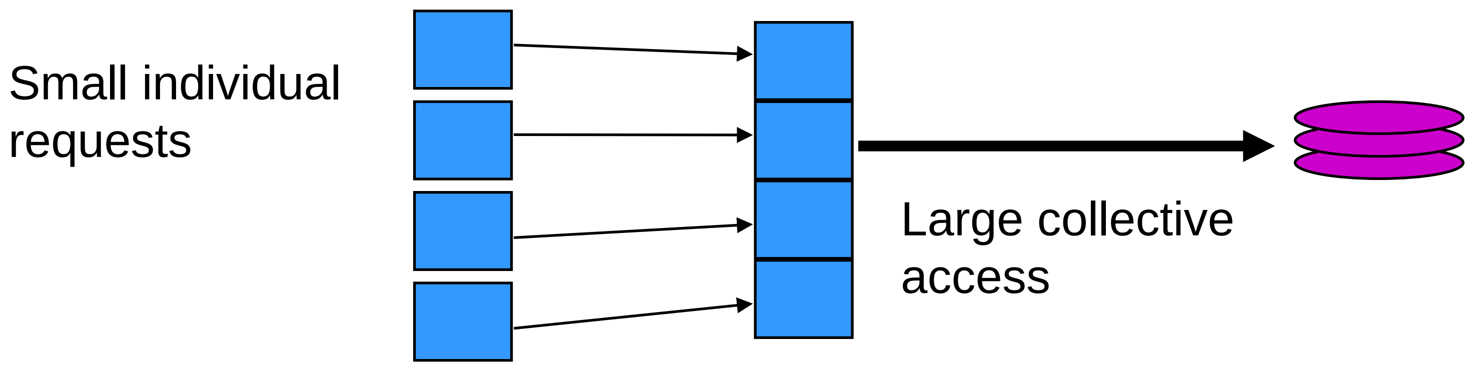
Code for Example #2

```
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



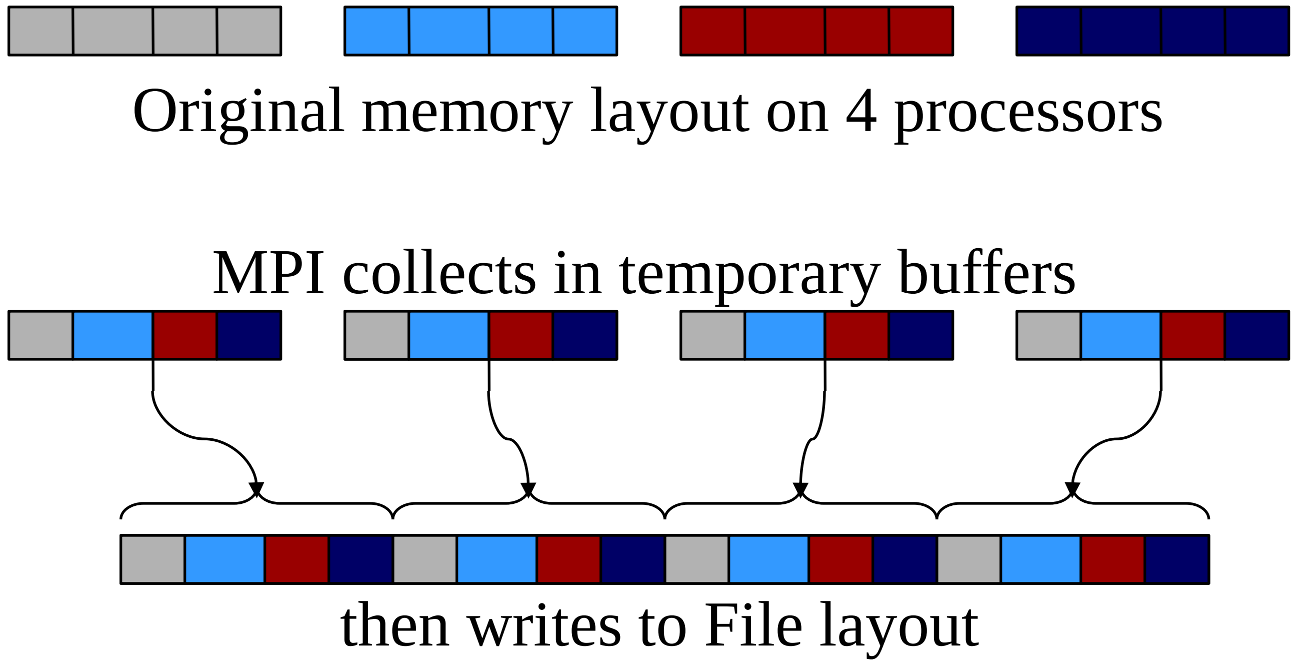
MPI Routines for Collective I/O

- 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`**

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.

Optional Lab

- Let’s run an MPI-IO program that writes in parallel to a single file and test how the speed depends on striping. First, compile the code.

```
tar xvfz ~tg459572/LABS/mpiio.tgz
cd mpiio; make
```

- Then examine ranger.sh. It performs the same striping commands you tried earlier. Here is what the script does:
  – Creates a working directory on $SCRATCH.
  – Copies mpiio writing and reading programs into that directory.
  – Runs the writing and reading test programs with default striping, taking timings in the process.
  – Repeats the tests for 8-way and 2-way striping.
  – Deletes the working directory.

Running the Optional Lab

- Submit ranger.sh with qsub. Don't forget to set the account to the correct account for this class.
- Some questions to ponder while waiting for the scheduler: what is the default stripe for $HOME, $WORK, and $SCRATCH? Do these choices make sense?
- After the job completes, you'll find the reading and writing rates for different stripe counts in the standard output that comes back from the job. Look for ====.
- Submit again and look for timing variability. If you like, you can change the BLOCKS variable to set a new size for the MPI-IO file prior to re-submitting.
- Credit: the MPI-IO program comes from http://beige.ucs.indiana.edu/I590/node86.html.