AMS 250: An Introduction to High Performance Computing

Parallel I/O

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

University of California, Santa Cruz

Outline

- Parallel I/O and Parallel File Systems
- Parallel I/O patterns
- MPI-IO
- HDF5
- Parallel HDF5

High Performance Computing and I/O

- High Performance Computing (HPC) applications often do I/O for
  - Reading initial conditions or datasets for processing
  - Writing numerical data from simulations
    - Parallel applications commonly need to write distributed arrays to disk
    - And to save **checkpoints** (application states that are written to disk for restarting the application)
- Efficient I/O without stressing out the HPC system is challenging
  - Load and store operations are more time-consuming than multiply operations
  - Total Execution Time = Computation Time + Communication Time + I/O Time
  - We need to optimize all the components of the equation above to get best performance

Relative Speed of Components in HPC

Xeon E5-2650	Latency (clocks)	Bandwidth	Size (bytes)
Registers	1	8192 GB/s	~100
L1 Cache	4	2048 GB/s	32 K (data) + 32K (instruction) per core
L2 Cache	11	744.7 GB/s	256 K per core
L3 Cache	25	327.7 GB/s	20 M shared
Local Memory	160	51.2 GB/s	~32GB
QPI	256	32.0 GB/s	
PCIe 2.0 x16	1024	8 GB/s	
InfiniBand QDR	2048	4 GB/s	
SSD	16384	~500 MB/s	~500GB
HDD	81920	~100 MB/s	~1 TB

The I/O gap between average disk access speed and memory speed stands at roughly 10^{-3}

File Systems

A file system is the software used to control how data is stored and retrieved. File system creates the abstractions of files, directories, access permissions, and so on.

- Local File Systems
  - Examples: ext2/3/4, XFS, Btrfs, ZFS, etc.
- Distributed File Systems
  - Designed to let processes on multiple computers access a common set of files
  - But *not* designed to give multiple processes efficient, concurrent access to the same file
  - Examples: NFS (Network File System), AFS, DFS, etc.
- Parallel File Systems

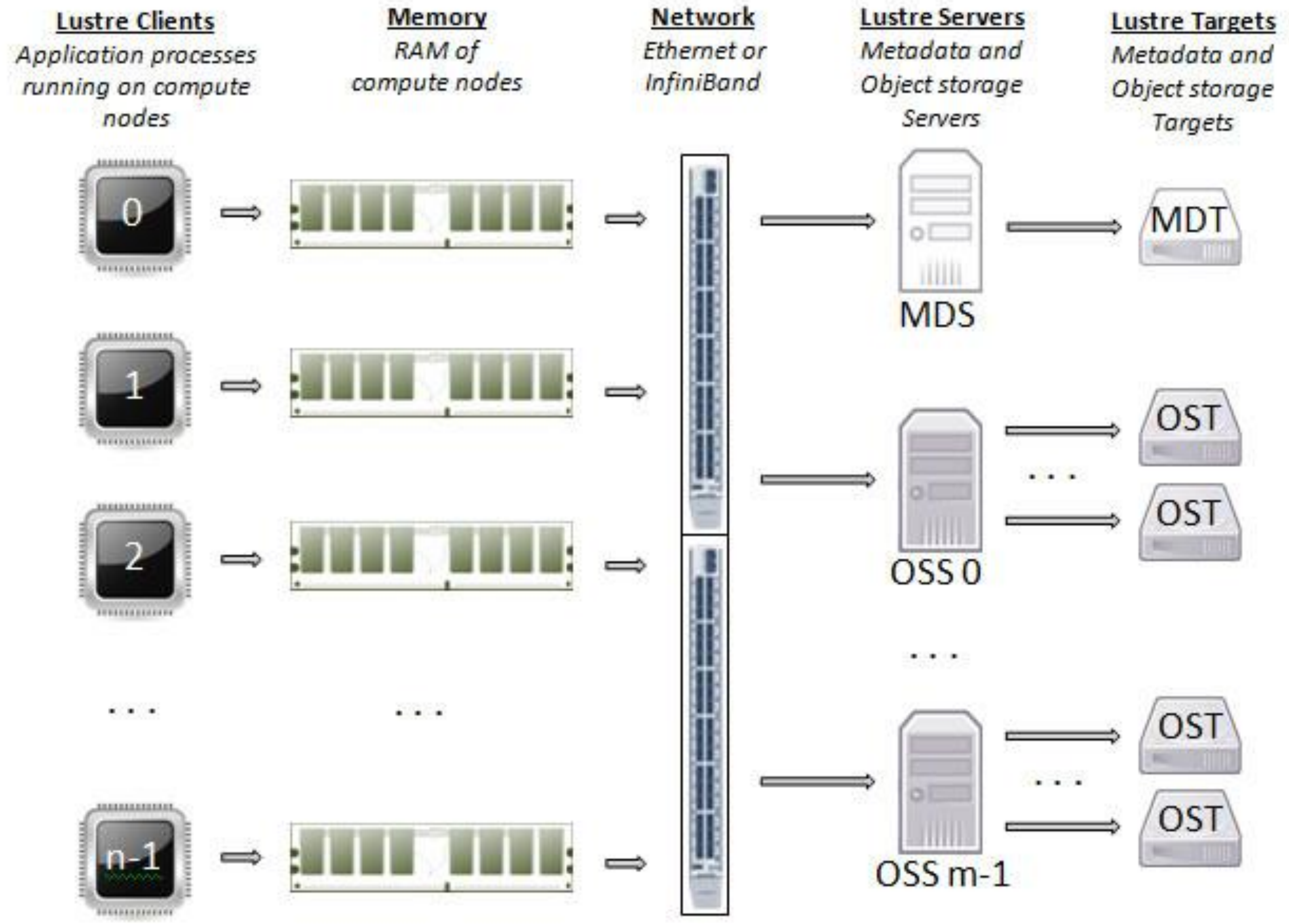
Parallel File Systems

- The design of a parallel file system must deal with several important questions:
  - How can thousands or millions of processes access the same file concurrently and efficiently
  - How should file pointers work?
  - Can the Unix sequential consistency semantics be preserved?
  - How can file blocks be cached and buffered?
- Examples:
  - Lustre: https://en.wikipedia.org/wiki/Lustre_(file_system)
  - GPFS: https://en.wikipedia.org/wiki/IBM_General_Parallel_File_System
  - BeeGFS: https://en.wikipedia.org/wiki/BeeGFS
  - PVFS: https://en.wikipedia.org/wiki/Parallel_Virtual_File_System
  - HDFS: https://en.wikipedia.org/wiki/Apache_Hadoop#HDFS

Lustre File System

- http://lustre.org/
- Lustre is a high-performance parallel file system, used by more than 60 of the top 100 fastest supercomputers in the world
- A Lustre file system has three major functional units:
  - One or more **metadata servers** (**MDS**es) that has one or more **metadata targets** (**MDT**s) per Lustre file system that stores namespace metadata, such as filenames, directories, access permissions, and file layout.
  - One or more **object storage servers** (**OSS**s) that store file data on one or more **object storage targets** (**OST**s). The capacity of a Lustre file system is the sum of the capacities provided by the OSTs.
  - Clients that access and use the data. Lustre presents all clients with a unified namespace for all of the files and data in the file system, using standard POSIX semantics, and allows concurrent and coherent read and write access to the files in the file system.

View of the Lustre File System



http://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips

File Striping

Lustre distributes the segments of a single file across multiple OSTs using a technique called **file striping**. A file is said to be **striped** when its linear sequence of bytes is separated into small chunks, or stripes, so that read and write operations can access multiple OSTs *concurrently*.

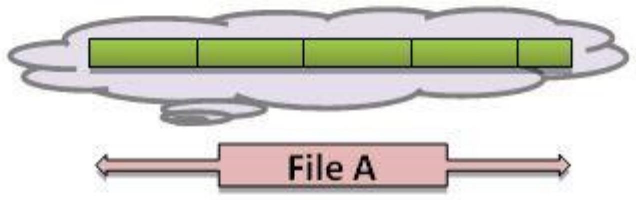
☺ an increase in the bandwidth available when accessing the file

☺ an increase in the available disk space for storing the file

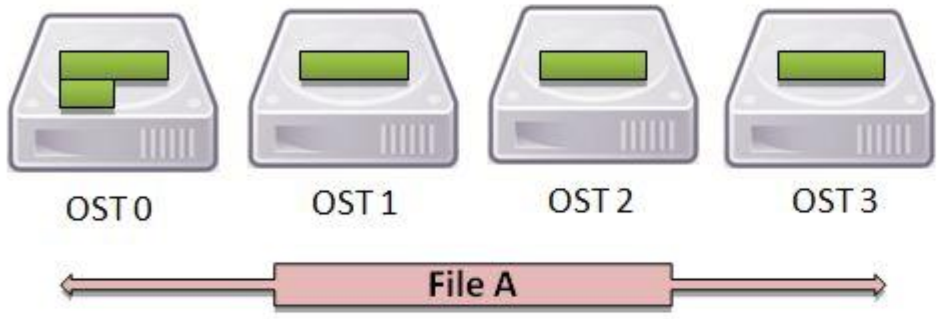
☹ increased overhead due to network operations and server contention

☹ increased risk of file damage due to hardware malfunction

NERSC Striping Shortcuts: http://www.nersc.gov/users/storage-and-file-systems/i-o-resources-for-scientific-applications/optimizing-io-performance-for-lustre/



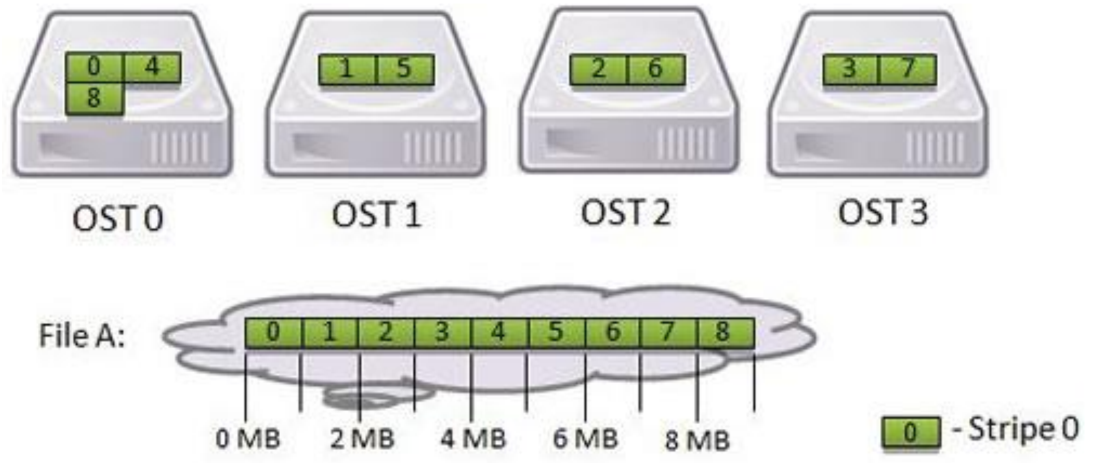
Logical view of a file



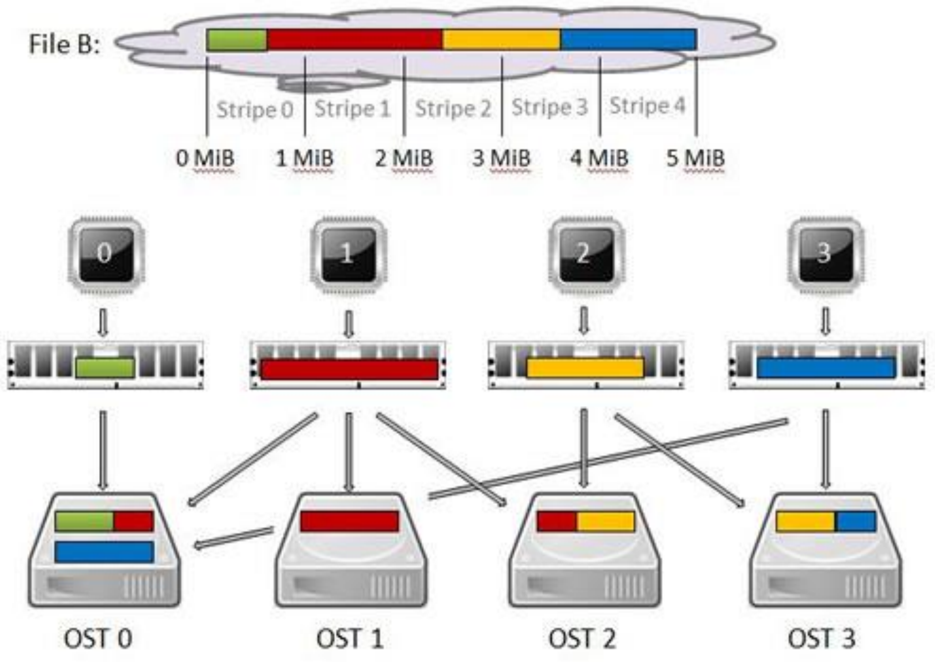
Physical view of a file

Stripe Alignment

Aligned Stripes



Non-aligned Striped



Performance can be improved, when:

1. processes access file locations that reside on different stripes
2. processes access the file at offsets which correspond to stripe boundaries

lfs utility

Lustre allows users to query or specify the striping policy for each file or directory of files using the **lfs** utility

- **lfs getstripe** lists the striping information for a file or directory. For examples:

```
$ lfs getstripe dir/file1
$ lfs getstripe dir
```

- Files and directories inherit striping patterns from the parent directory. However, you can change them for a single file, multiple files, or a directory using the **lfs setstripe** command. For examples:
  - To create a *new* zero length file named *file1* with a stripe size of 2MB (default is 1MB) and a stripe count of 8:

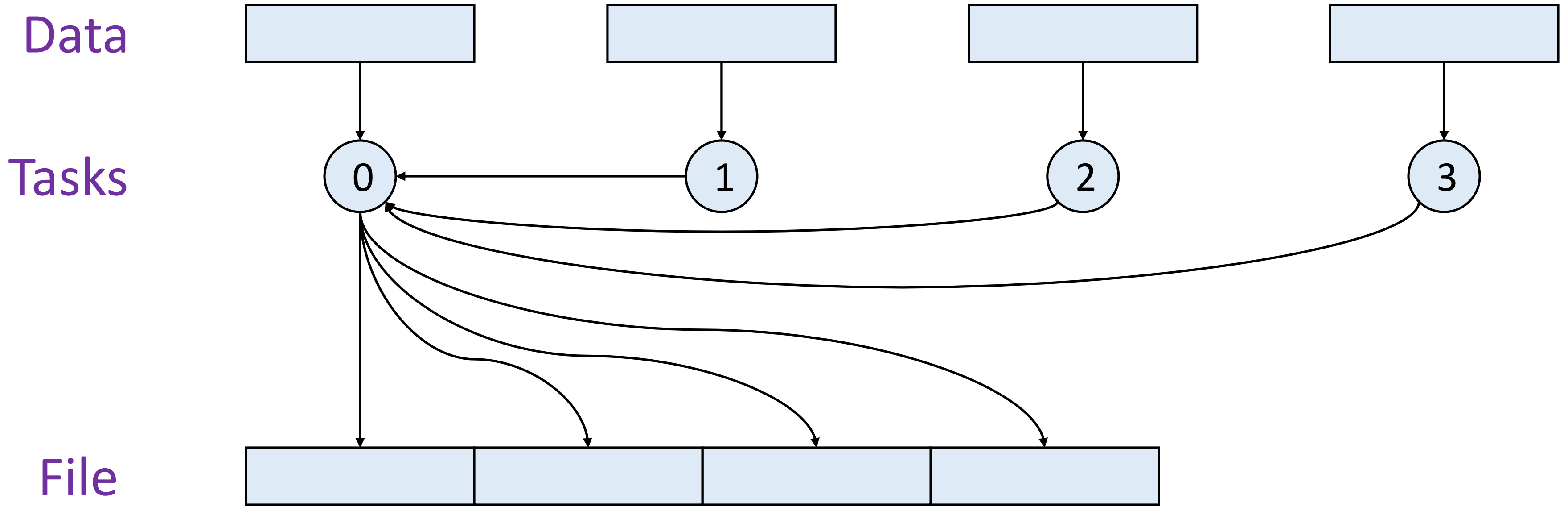
```
$ lfs setstripe file1 -s 2m -c 8
```

  - To set a default striping configuration for any *new* files created in the directory *dir1* (existing files in the directory are not affected):

```
$ lfs setstripe dir1 -c 8
```

Typical Pattern: Sequential I/O

All processes send data to master process; then the master writes the collected data to the file

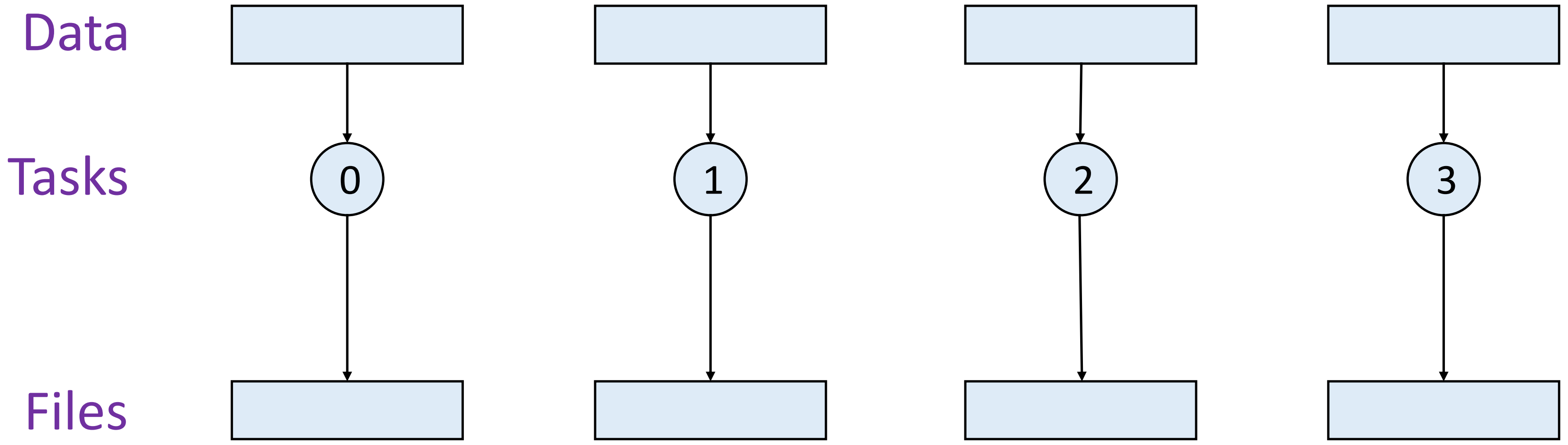


Pros:
- Simple

Cons:
- Scales poorly
- May not fit into memory on task 0
- Bandwidth from 1 task is very limited

Typical Pattern: Parallel I/O Multi-file

Each process writes to a separate file



Pros:

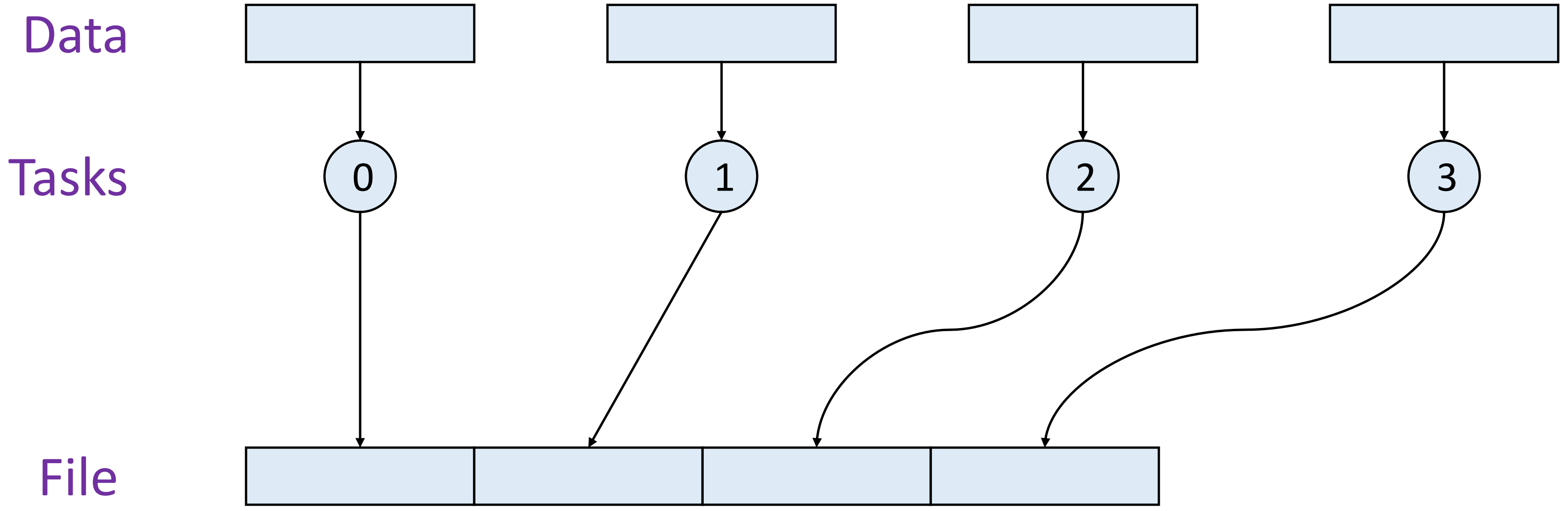
- Easy to program
- Can be fast (up to a point)

Cons:

- Difficult to manage a lot of small files
- Many files can cause serious performance problems to any file system

Typical Pattern: Parallel I/O Single-file

Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



Pros:

- High performance
- Single file makes data manageable

Cons:

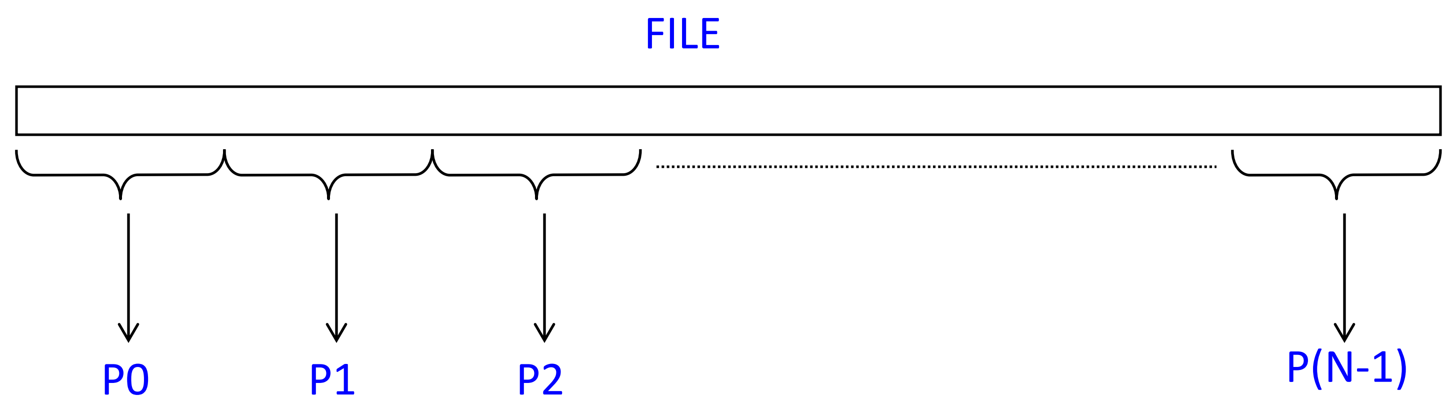
- Can be more difficult to program

MPI for Parallel I/O

- A parallel I/O system for distributed-memory architecture will need a mechanism to
  - define collective operations – *MPI communicators*
  - define noncontiguous data layout in memory and file – *MPI datatypes*
  - test completion of nonblocking operations – *MPI request objects*
- Reading and writing are like receiving and sending messages
- Hence, an MPI-like machinery is a good setting for Parallel I/O
- MPI-IO is part of the MPI-2 standard, released in July 1997

Using MPI-IO

- 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 (e.g., **MPI_File_read_at**)
  - Access a shared file pointer (e.g., **MPI_File_seek_shared**, **MPI_File_read_shared**)



Opening a File

- Calls to MPI functions for reading or writing must be preceded by a call to **MPI_File_open**

C	`int MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)`
Fortran	`MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)` `CHARACTER*(*) FILENAME` `INTEGER COMM, AMODE, INFO, FH, IERROR`

- The following access mode are supported:

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
- Collective function

Info Object

- Programmer may wish to provide additional information to MPI, in hope that MPI would know what to do with it.
- Such information is referred to as *hints* and there is a special MPI construct called the **info** object that is supposed to collect all the *hints*.
- **info** is an opaque object with a handle of type **MPI_Info** in C, and **INTEGER** in Fortran.
- It stores an unordered set of (key, value) pairs (both *key* and *value* are strings).
- An implementation must support *info* objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key.
- **MPI_File_get_info** used to get list of *hints* supported by the implementation.

Passing Hints to the MPI Implementation

```
MPI_File fh;

MPI_Info info;
MPI_Info_create(&info);

/* set Lustre stripe count */
MPI_Info_set(info, "striping_factor", "4");
/* set Lustre stripe size in bytes (2M) */
MPI_Info_set(info, "striping_unit", “2097152");

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

MPI_Info_free(&info);
```

Closing a File

- Close the file using **MPI_File_open**
- Collective function

C	`int MPI_File_close(MPI_File *fh)`

Fortran	`MPI_FILE_CLOSE(FH, IERROR)`
	`    INTEGER    FH, IERROR`

https://www.open-mpi.org/doc/current/

Reading Files

After opening a file, read data from the file by either using **MPI_File_seek** & **MPI_File_read** or **MPI_File_read_at**

C	`int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)`

Fortran	`MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)`
	`INTEGER FH, WHENCE, IERROR`
	`INTEGER(KIND=MPI_OFFSET_KIND) OFFSET`

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

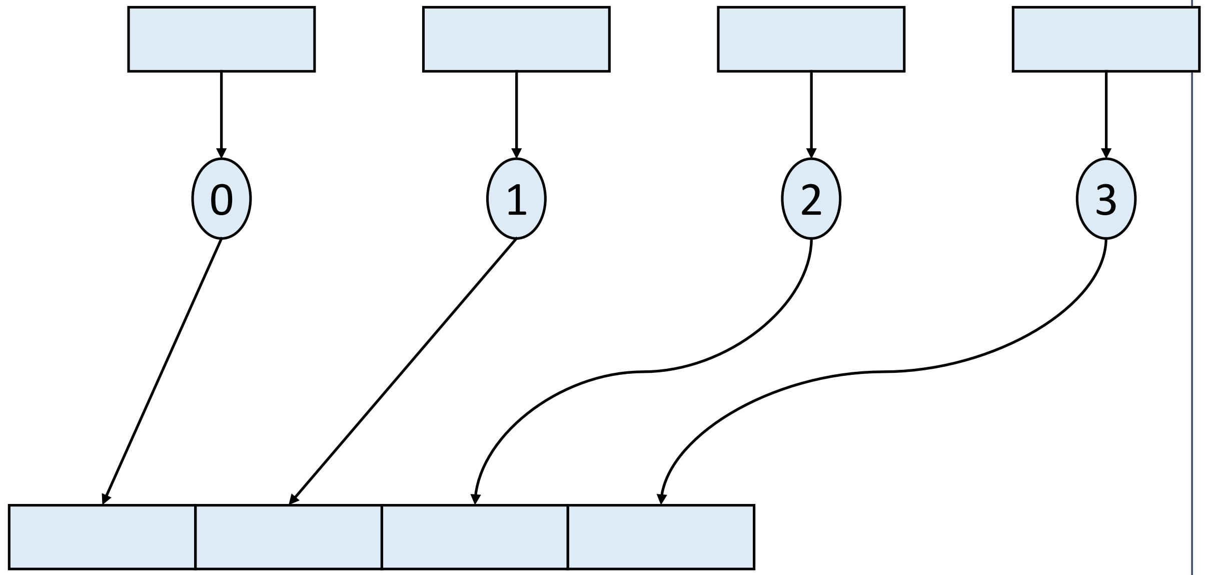
Fortran	`MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)`
	`<type>     BUF(*)`
	`INTEGER    FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR`

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

Fortran	`MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)`
	`<type>    BUF(*)`
	`INTEGER   FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR`
	`INTEGER(KIND=MPI_OFFSET_KIND) OFFSET`

Reading a File: readFile1.c

```
/* read from a common file using individual file pointers */
#include "mpi.h"
#define FILESIZE (1024 * 1024)
int main(int argc, char **argv){
  int *buf, 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;
  buf = (int *) malloc(bufsize);
  nints = bufsize/sizeof(int);
  MPI_File_open(MPI_COMM_WORLD, "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);
  free(buf);
  MPI_Finalize();
  return 0;
}
```



Reading a File: readFile2.f90

```
! read from a common file using explicit offsets
PROGRAM main
  use mpi
  integer FILESIZE, MAX_BUFSIZE, INTSIZE
  parameter (FILESIZE=1048576, MAX_BUFSIZE=1048576, INTSIZE=4)
  integer buf(MAX_BUFSIZE), rank, ierr, fh, size, nints
  integer status(MPI_STATUS_SIZE)
  integer (kind=MPI_OFFSET_KIND) offset
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
  call MPI_FILE_OPEN(MPI_COMM_WORLD, 'datafile', &
                     MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
  nints = FILESIZE/(size*INTSIZE)
  offset = rank * nints * INTSIZE
  call MPI_FILE_READ_AT(fh, offset, buf, nints, MPI_INTEGER, status, ierr)
  call MPI_FILE_CLOSE(fh, ierr)
  call MPI_FINALIZE(ierr)
END PROGRAM main
```

Writing Files

- While opening a file in the write mode, use the appropriate flag(s) in **MPI_File_open**: MPI_MODE_WRONLY or MPI_MODE_RDWR and MPI_MODE_CREATE if needed
- For writing files, use **MPI_File_set_view** & **MPI_File_write** or **MPI_File_write_at**

C	`int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,`
	`MPI_Datatype filetype, const char *datarep, MPI_Info info)`
Fortran	`MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)`
	`INTEGER FH, ETYPE, FILETYPE, INFO, IERROR`
	`CHARACTER*(*) DATAREP`
	`INTEGER(KIND=MPI_OFFSET_KIND) DISP`

C	`int MPI_File_write(MPI_File fh, const void *buf, int count,`
	`MPI_Datatype datatype, MPI_Status *status)`
Fortran	`MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)`
	`<type>     BUF(*)`
	`INTEGER    FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR`

C	`int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,`
	`int count, MPI_Datatype datatype, MPI_Status *status)`
Fortran	`MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)`
	`<type>    BUF(*)`
	`INTEGER   FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR`
	`INTEGER(KIND=MPI_OFFSET_KIND) OFFSET`

Writing a File: writeFile1.c

```
/* write to a common file using explicit offsets */
#include "mpi.h"
#define FILESIZE (1024 * 1024)
int main(int argc, char **argv) {
  int *buf, i, rank, size, bufsize, nints, offset;
  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;
  buf = (int *) malloc(bufsize);
  nints = bufsize/sizeof(int);
  for (i=0; i<nints; i++) buf[i] = rank*nints + i;
  offset = rank*bufsize;
  MPI_File_open(MPI_COMM_WORLD, "datafile", MPI_MODE_CREATE|MPI_MODE_WRONLY,
                MPI_INFO_NULL, &fh);
  MPI_File_write_at(fh, offset, buf, nints, MPI_INT, &status);
  MPI_File_close(&fh); free(buf);
  MPI_Finalize();
  return 0;
}
```

Using File Views for Writing a Shared File

- When processes need to write to a shared file, assigns regions of the file to separate processes using **MPI_File_set_view**

C	`int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)`
Fortran	`MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)` `INTEGER FH, ETYPE, FILETYPE, INFO, IERROR` `CHARACTER*(*) DATAREP` `INTEGER(KIND=MPI_OFFSET_KIND) DISP`

- File view is 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 processes

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

File Data Representation

- **native**
  - data is stored in the file as it is in memory; no data conversion is performed
- **internal**
  - an implementation-defined representation that may provide some (implementation-defined) degree of file portability
- **external32**
  - a specific data representation defined in MPI
  - basically a 32-bit big-endian IEEE format, with the sizes of all basic data types specified by MPI
  - A file written with external32 can be read with any MPI implementation on any machine
  - Since using external32 may require the implementation to perform data conversion, it may result in lower I/O performance and some loss in data precision

Writing a File: writeFile2.f90

```
! write to a common file using file view
PROGRAM main
  use mpi
  integer FILESIZE, ierr, i, rank, size, intsize, nints, fh
  parameter (FILESIZE=1048756)
  integer buf(FILESIZE)
  integer(kind=MPI_OFFSET_KIND) disp
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
  call MPI_TYPE_SIZE(MPI_INTEGER, intsize, ierr)
  nints = FILESIZE/(size*intsize)
  do i = 1, nints
    buf(i) = rank * nints + i - 1
  enddo
  call MPI_FILE_OPEN(MPI_COMM_WORLD, 'datafile', &
                     MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)
  disp = rank * nints * intsize
  call MPI_FILE_SET_VIEW(fh, disp, MPI_INTEGER, MPI_INTEGER, 'native', &
                         MPI_INFO_NULL, ierr)
  call MPI_FILE_WRITE(fh, buf, nints, MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
  call MPI_FILE_CLOSE(fh, ierr)
  call MPI_FINALIZE(ierr)
END PROGRAM main
```

Try them out on Cori Phase I

```
cori03:> cc readFile1.c -o readFile1.x
cori03:> ftn readFile2.f90 -o readFile2.x
cori03:> cc writeFile1.c -o writeFile1.x
cori03:> ftn writeFile2.f90 -o writeFile2.x

cori03:> salloc -N 1 -p debug -L SCRATCH -C haswell

nid00468:> srun -n 2 ./writeFile1.x
nid00468:> srun -n 4 ./readFile2.x
nid00468:> rm datafile
nid00468:> srun -n 4 ./writeFile2.x
nid00468:> srun -n 2 ./readFile1.x
```

Try them out on Hyades (Intel MPI)

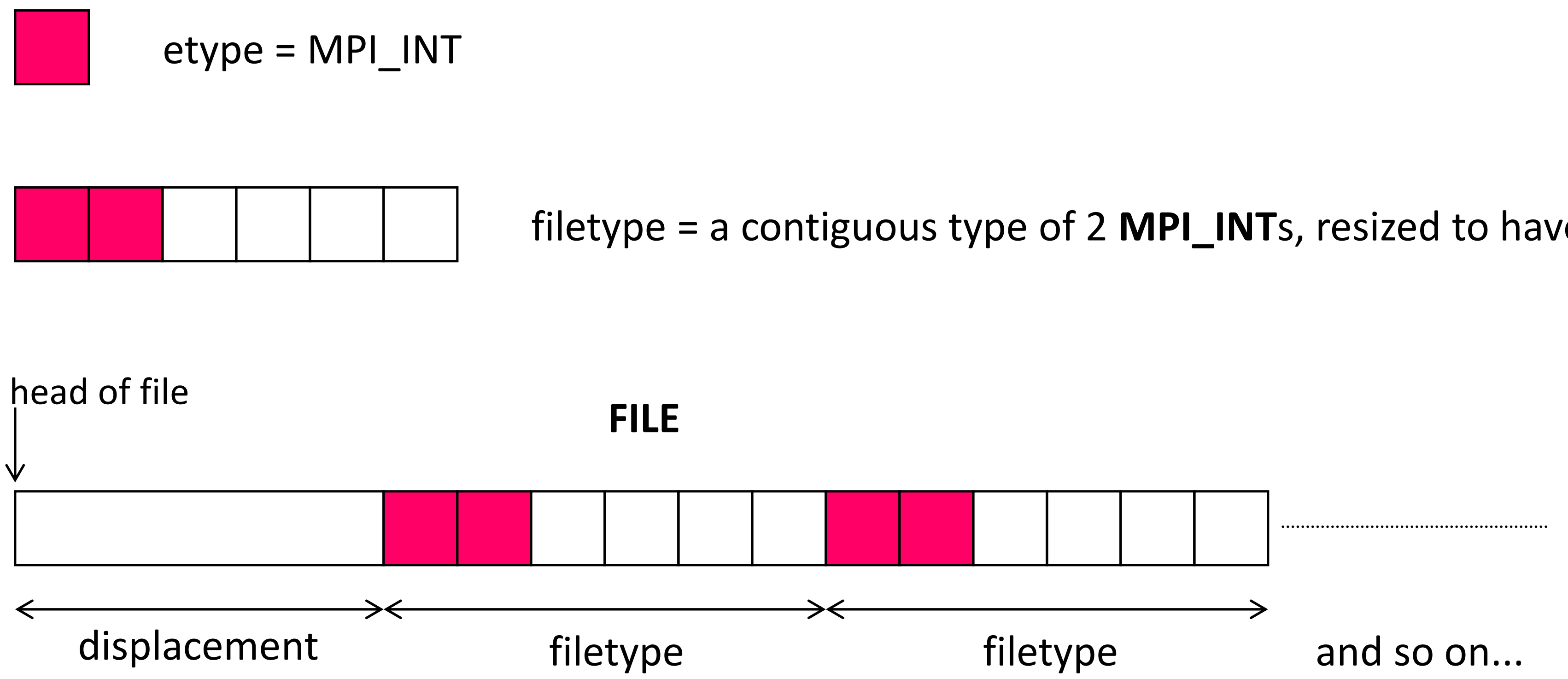
```
$ mpiicc readFile1.c -o readFile1.x
$ mpiifort readFile2.f90 -o readFile2.x
$ mpiicc writeFile1.c -o writeFile1.x
$ mpiifort writeFile2.f90 -o writeFile2.x

$ export I_MPI_EXTRA_FILESYSTEM=on
$ export I_MPI_EXTRA_FILESYSTEM_LIST=lustre

$ mpirun -n 2 ./writeFile1.x
$ mpirun -n 4 ./readFile2.x
$ rm datafile
$ mpirun -n 4 ./writeFile2.x
$ mpirun -n 2 ./readFile1.x
```

http://www.mcs.anl.gov/projects/romio/2014/06/12/romio-and-intel-mpi/

A File View Example



etype = MPI_INT

filetype = a contiguous type of 2 **MPI_INT**s, resized to have an extent of 6 **MPI_INT**s

Once this view is set , only the shaded portions of the file will be read/written by any read/write function; the blank unshaded portions will be skipped.

Sample File View Code

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;
MPI_File fh;
int buf[1000];

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0;
extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int);
etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Note on Atomic Read/Write

C	`int MPI_File_set_atomicity(MPI_File fh, int flag)`

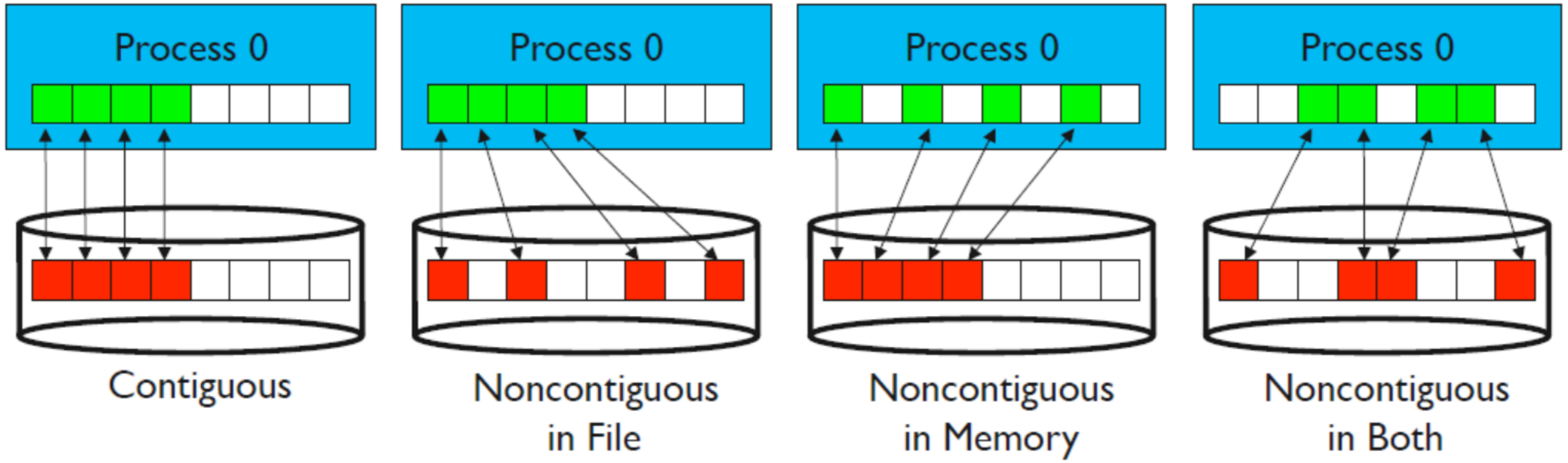
Fortran	`MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)`
	`INTEGER        FH, FLAG, IERROR`

- Use this API to set the atomic 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
- Collective function

Collective I/O in MPI

- 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 and interleaved

Contiguous and Noncontiguous I/O



- **Contiguous I/O** moves data from a single memory block into a single file region
- **Noncontiguous I/O** has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g., block decomposition)
- Describing noncontiguous accesses with a single operation (collective I/O) passes more knowledge to I/O system

Collective I/O Functions

C	`int` **`MPI_File_read_all`**`(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
Fortran	**`MPI_FILE_READ_ALL`**`(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)` `<type>     BUF(*)` `INTEGER    FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR`

C	`int` **`MPI_File_write_all`**`(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
Fortran	**`MPI_FILE_WRITE_ALL`**`(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)` `<type>     BUF(*)` `INTEGER    FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR`

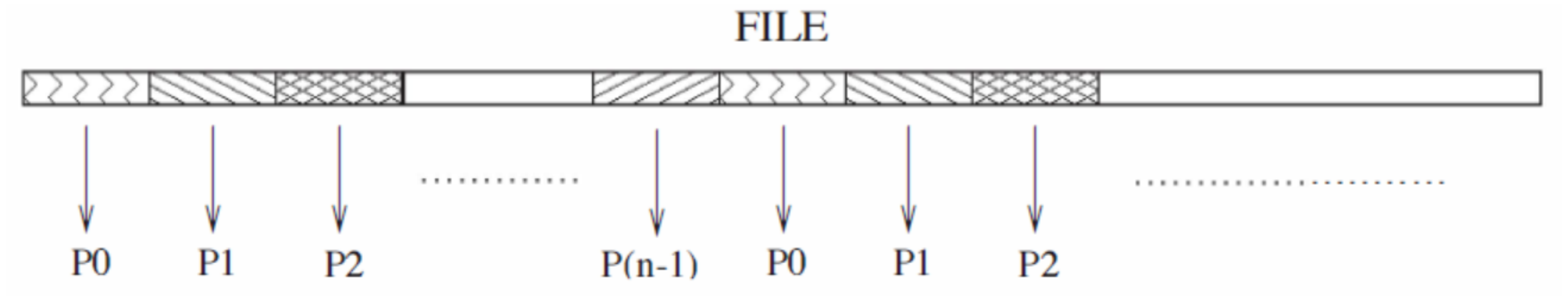
C	`int` **`MPI_File_read_at_all`**`(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
Fortran	**`MPI_FILE_READ_AT_ALL`**`(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)` `<type>     BUF(*)` `INTEGER    FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR` `INTEGER(KIND=MPI_OFFSET_KIND) OFFSET`

C	`int` **`MPI_File_write_at_all`**`(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
Fortran	**`MPI_FILE_WRITE_AT_ALL`**`(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)` `<type>     BUF(*)` `INTEGER    FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR` `INTEGER(KIND=MPI_OFFSET_KIND) OFFSET`

Collective I/O Example

An example of *collective* I/O together with *noncontiguous* accesses:

Each process reads blocks of data distributed in a *round-robin* (block-cyclic) manner in the file

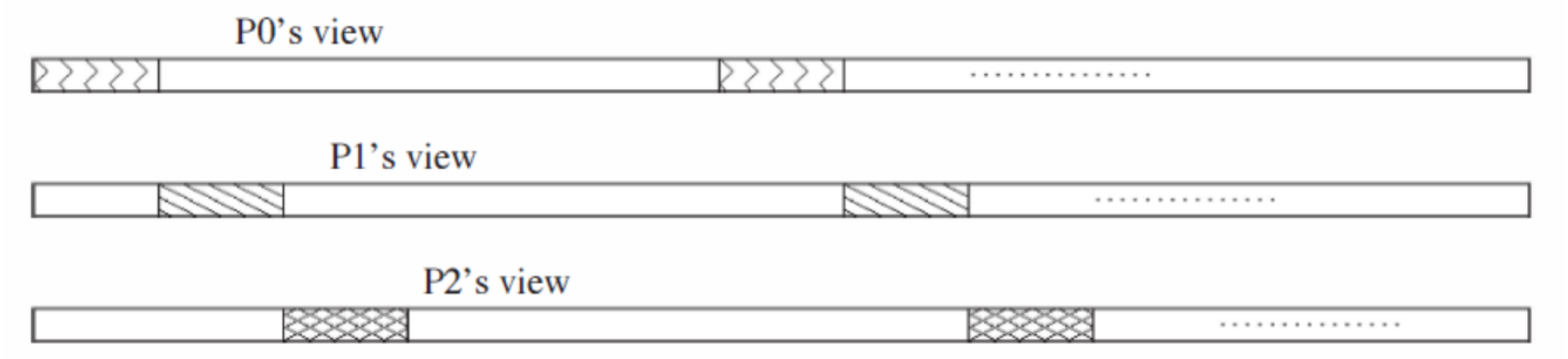


collectiveRead.c

```
/* noncontiguous access with a single collective I/O function */
#include "mpi.h"
#define FILESIZE 1048576
#define INTS_PER_BLK 16
int main (int argc, char **argv)
{
  int *buf, rank, size, nints, bufsize;
  MPI_File fh ;
  MPI_Datatype filetype;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  buf = (int *) malloc(bufsize);
  nints = bufsize/sizeof(int);
  MPI_File_open(MPI_COMM_WORLD, "datafile", MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
  MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK,
                  INTS_PER_BLK*size, MPI_INT, &filetype);
  MPI_Type_commit(&filetype);
```

collectiveRead.c (cont'd)

```
  MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT,
                    filetype, "native", MPI_INFO_NULL);
  MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
  MPI_File_close(&fh);
  MPI_Type_free(&filetype);
  free(buf);
  MPI_Finalize();
  return 0 ;
}
```



Nonblocking I/O

```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count, datatype, &request);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_Wait(&request, &status);
```

Split Collective I/O

- A restricted form of nonblocking collective I/O
- Only one active nonblocking collective operation allowed at a time on a file handle
- Therefore, no request object necessary

```
MPI_File_write_all_begin(fh, buf, count, datatype);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_File_write_all_end(fh, buf, &status);
```

Shared File Pointers

- 3 ways of specifying to MPI the location in the file from where data must be read/written:
  - individual file pointers
  - explicit offsets
  - shared file pointers
- Shared file pointer is shared among the processes belonging to the communicator passed to **MPI_File_open**
- After a call to **MPI_File_read_shared** or **MPI_File_write_shared**, the shared file pointer is updated by the amount of data read or written
- The next call to one of these functions from any process in the group will result in data being read or written from the new location of the shared file pointer.
- A process can explicitly move the shared file pointer (in units of etypes) by using the function **MPI_File_seek_shared**

Shared File Pointer Example

```
/* write to a common file using shared file pointer */
#include "mpi.h"
int main(int argc, char **argv) {
  int buf[1000];
  MPI_File fh;
  MPI_Init(&argc, &argv);
  MPI_File_open(MPI_COMM_WORLD, "datafile", MPI_MODE_CREATE|MPI_MODE_WRONLY,
                MPI_INFO_NULL, &fh);
  MPI_File_write_shared(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Guidelines for Achieving High I/O Performance

- Use fast file systems (e.g., Lustre), not NFS-mounted home directories
- Do not perform I/O from one process only
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call

Common Storage Formats

- ASCII
  - Very slow
  - Takes a lot of space!
  - Inaccurate
- Binary
  - Non-portable (e.g., byte ordering and types sizes)
  - Not future proof
- Self-describing formats
  - NetCDF, HDF5, Parallel NetCDF, etc.
- Community file formats
  - FITS, HDF-EOS, SAF, PDB, Plot3D, etc.
  - Modern Implementations built on top of HDF, NetCDF, or other self-describing object-model API

Higher Level I/O Interfaces

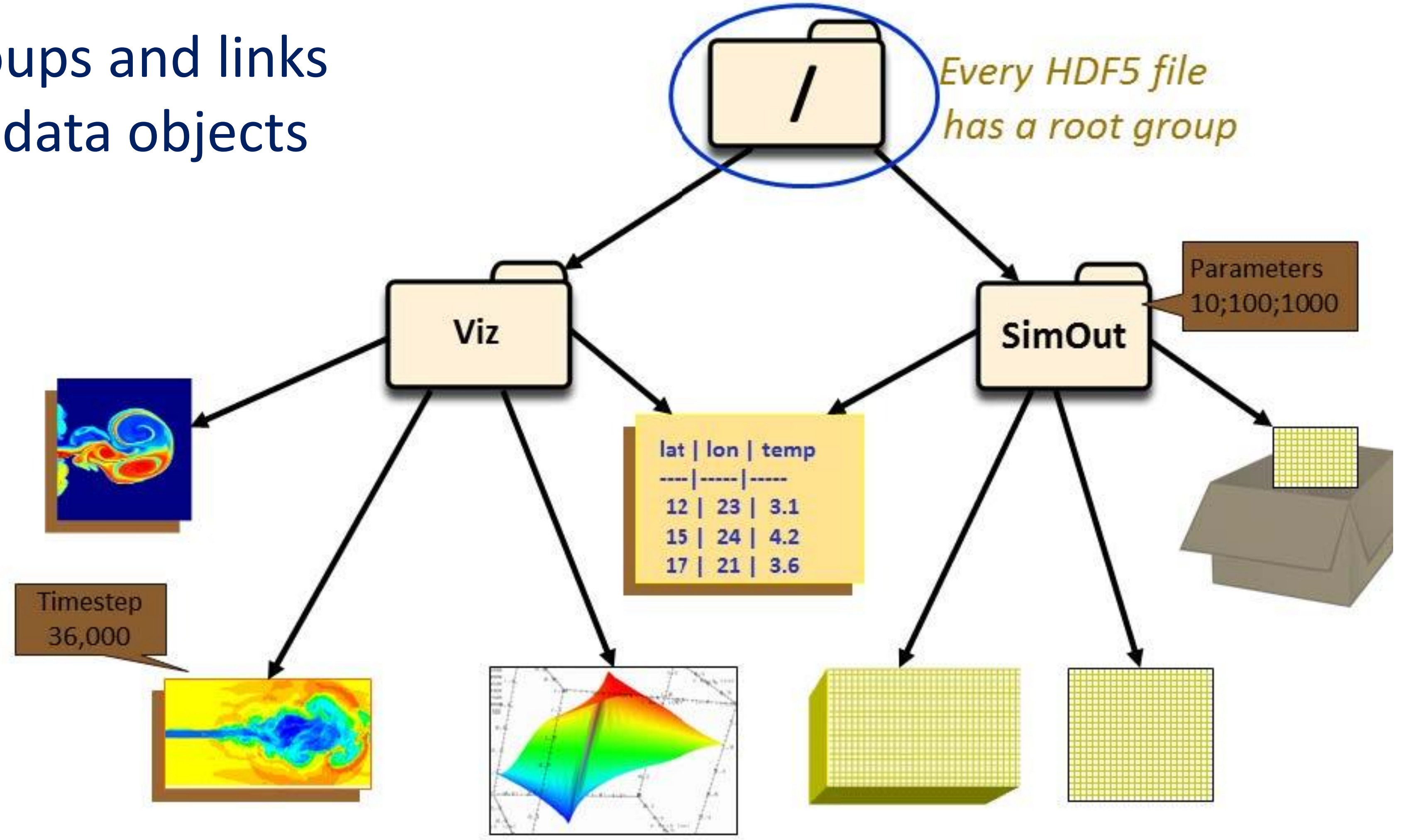
- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents
- Present APIs more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays
- Both Parallel HDF5 and Parallel netCDF are implemented on top of MPI-IO

HDF5

- https://www.hdfgroup.org/HDF5/
- The **HDF5** (Hierarchical Data Format 5) technology suite includes:
  - A versatile *data model* that can represent very complex data objects and a wide variety of metadata
  - A completely *portable file format* with no limit on the number or size of data objects in the collection
  - A software library that runs on a range of computational platforms, from laptops to massively parallel systems, and implements a high-level API with C, C++, Fortran 90, and Java interfaces
  - A rich set of integrated performance features that allow for access time and storage space optimizations
  - Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection

HDF5 File is a Container of Objects

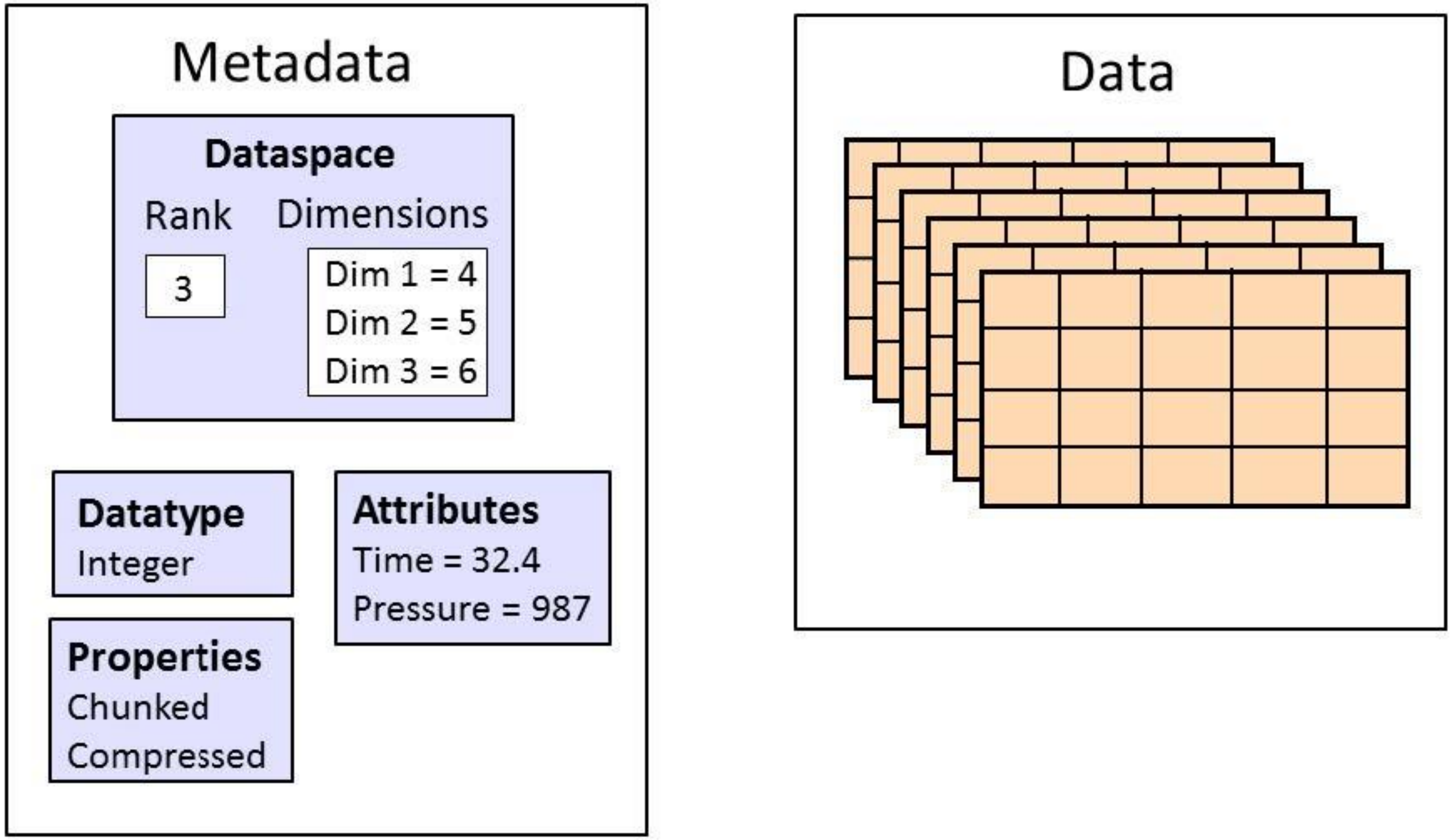
HDF5 groups and links **organize** data objects



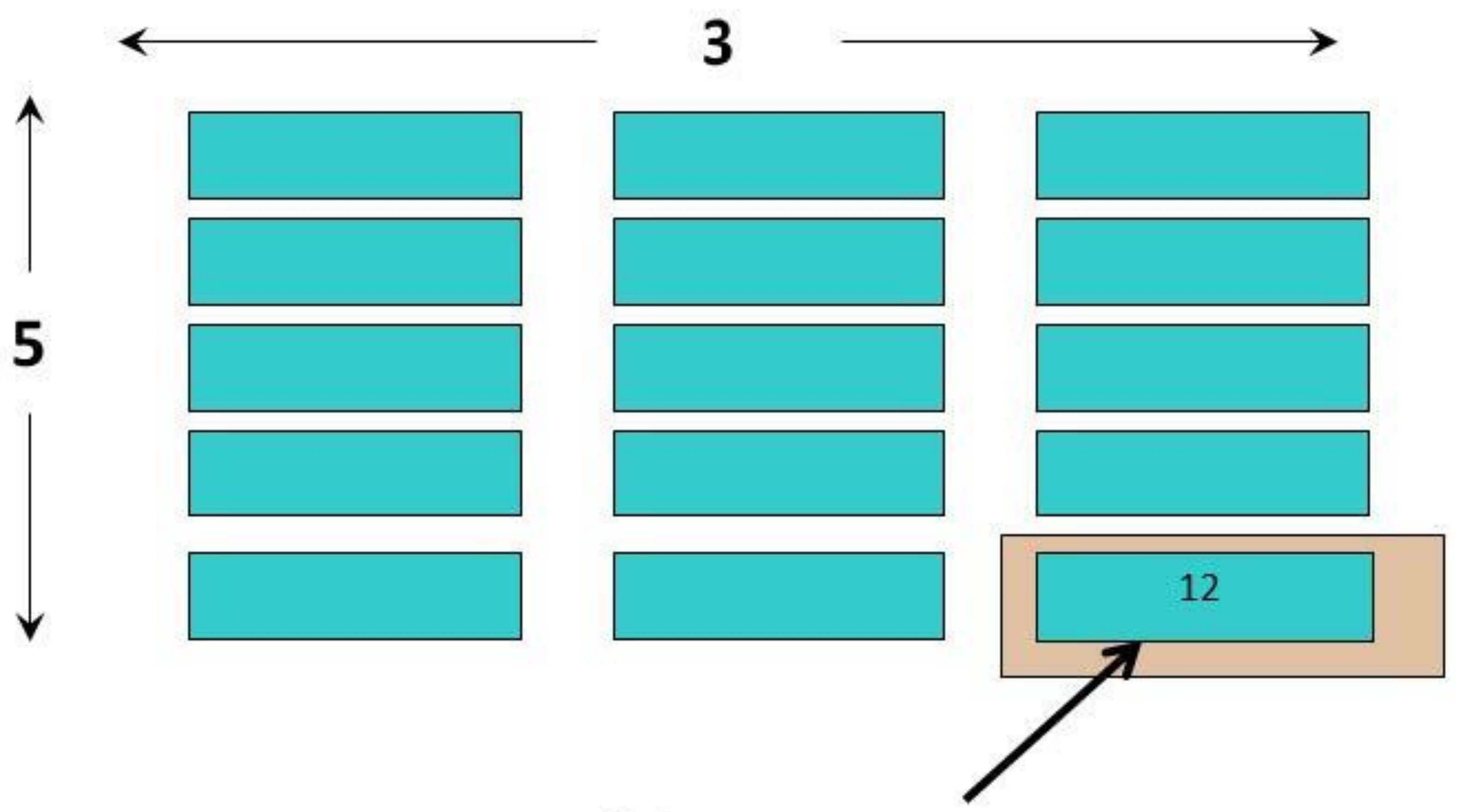
```
lat | lon | temp
----|-----|-----
 12 | 23 | 3.1
 15 | 24 | 4.2
 17 | 21 | 3.6
```

https://www.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf

HDF5 Dataset



HDF5 Dataset

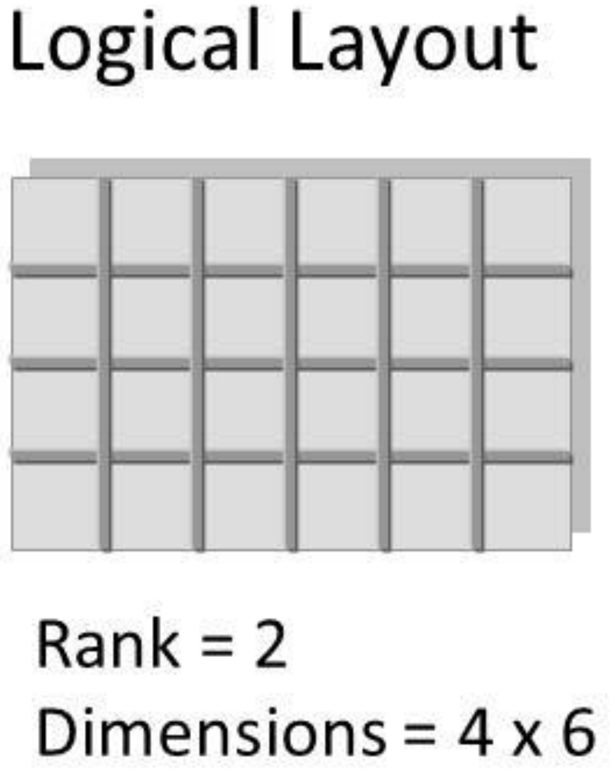


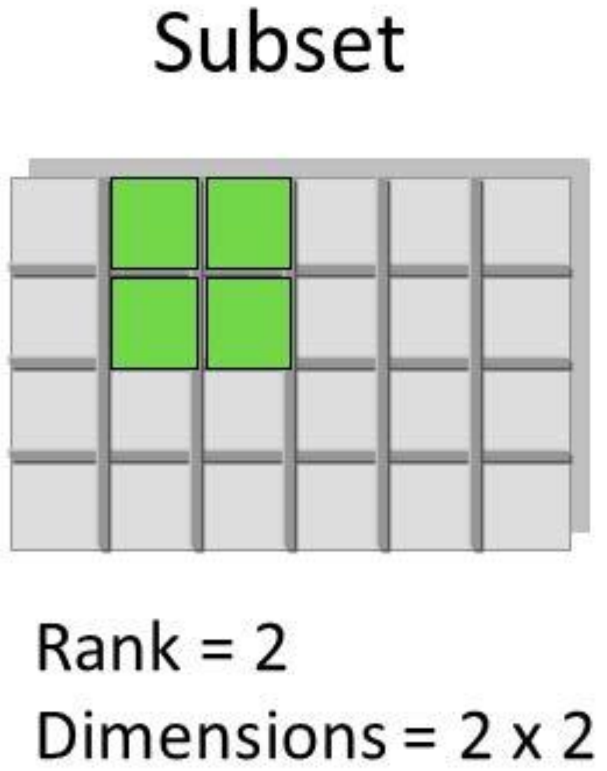
Datatype: 32-bit Integer

Dataspace: Rank = 2

Dimension = 5 x 3

HDF5 Dataspace





The dataspace is used to describe both the logical layout of a dataset and a subset of a dataset.

HDF5 Datatypes

The HDF5 datatype describes how to interpret individual data element.

HDF5 datatypes include:

- integer, float, unsigned, bitfield, …
- user-definable (e.g., 13-bit integer)
- variable length types (e.g., strings)
- references to objects/dataset regions
- enumerations - names mapped to integers
- opaque
- compound (similar to C structs)

HDF5 Pre-defined Datatype Identifiers

HDF5 defines a set of Datatype Identifiers per HDF5 session.

For example:

C Type	HDF5 File Type	HDF5 Memory Type
int	H5T_STD_I32BE H5T_STD_I32LE	H5T_NATIVE_INT
float	H5T_IEEE_F32BE H5T_IEEE_F32LE	H5T_NATIVE_FLOAT
double	H5T_IEEE_F64BE H5T_IEEE_F64LE	H5T_NATIVE_DOUBLE

HDF5 Defined Types

For portability, the HDF5 library has its own defined types:

hid_t:	object identifiers (native *integer*)
hsize_t:	size used for dimensions (*unsigned long* or *unsigned long long*)
herr_t:	function return value

For **C**, add "#include hdf5.h" in your HDF5 application;

For **Fortran**, add "USE HDF5" in your HDF5 application.

HDF5 APIs and Libraries

There are APIs for each type of object in HDF5. For example, all **C** routines in the HDF5 library begin with a prefix of the form **H5***, where * is one or two uppercase letters indicating the type of object on which the function operates:

H5A	**A**ttribute Interface
H5D	**D**ataset Interface
H5F	**F**ile Interface
H5G	**G**roup Interface
H5L	**L**ink Interface
H5O	**O**bject Interface
H5P	**P**roperty List Interface
H5S	Data**S**pace Interface
H5T	Data**T**ype Interface

Similarly the **FORTRAN** wrappers come in the form of subroutines that begin with **h5** and end with **_f**.

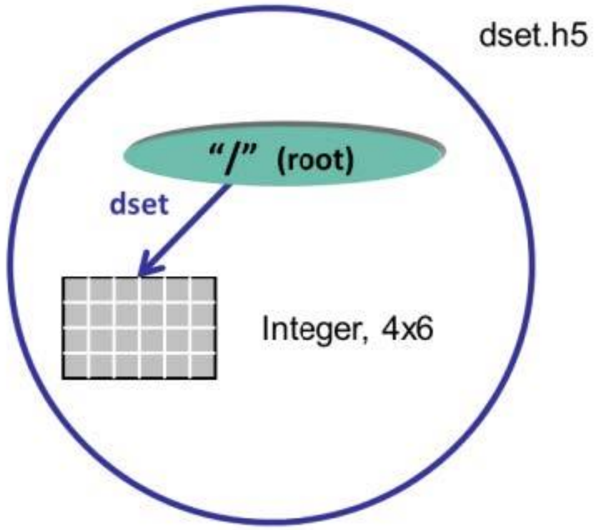
Basic Functions

`H5Fcreate (H5Fopen)`	*create (open) File*
`H5Screate_simple/H5Screate`	*create fileSpace*
`H5Dcreate (H5Dopen)`	*create (open) Dataset*
`H5Sselect_hyperslab`	*select subsections of data*
`H5Dread, H5Dwrite`	*access Dataset*
`H5Dclose`	*close Dataset*
`H5Sclose`	*close fileSpace*
`H5Fclose`	*close File*

NOTE: Order not strictly specified.

A Simple Example

1. Create an HDF5 file
2. Create a dataset
3. Write to the dataset
4. Read from the dataset
5. Close the dataset handle
6. Close the HDF5 file



https://www.hdfgroup.org/HDF5/examples/intro.html

C Code: h5_rdwt.c

```c
#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;
   int          i, j, dset_data[4][6];

   /* Create a new file using default properties */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the dataspace for the dataset */
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset */
   dataset_id = H5Dcreate2(file_id, "/dset", H5T_STD_I32BE, dataspace_id,
                           H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
```

C Code: h5_rdwt.c (cont'd)

```
    /* Initialize the dataset. */
    for (i = 0; i < 4; i++)
       for (j = 0; j < 6; j++)
          dset_data[i][j] = i * 6 + j + 1;

    /* Write the dataset */
    status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                      dset_data);
    /* Read the dataset */
    status = H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                     dset_data);
    /* End access to the dataset and release resources used by it */
    status = H5Dclose(dataset_id);

    /* Terminate access to the dataspace */
    status = H5Sclose(dataspace_id);

    /* Close the file */
    status = H5Fclose(file_id);
}
```

Try it out on Hyades

```
$ h5cc -o h5_rdwt.x h5_rdwt.c
$ h5cc -show
gcc -D_LARGEFILE64_SOURCE -D_LARGEFILE_SOURCE -L/usr/lib64 -lhdf5_hl -lhdf5 \
    -lrt -lsz -lz -ldl -lm -Wl,-rpath -Wl,/usr/lib64
```

or

```
$ gcc -o h5_rdwt.x h5_rdwt.c -lhdf5
$ ./h5_rdwt.x
$ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 1, 2, 3, 4, 5, 6,
      (1,0): 7, 8, 9, 10, 11, 12,
      (2,0): 13, 14, 15, 16, 17, 18,
      (3,0): 19, 20, 21, 22, 23, 24
      }
   }
}}
```

Fortran Code: h5_rdwt.f90

```fortran
PROGRAM H5_RDWT

  USE HDF5

  IMPLICIT NONE

  CHARACTER(LEN=8), PARAMETER :: filename = "dsetf.h5" ! File name
  CHARACTER(LEN=4), PARAMETER :: dsetname = "dset"     ! Dataset name

  INTEGER(HID_T) :: file_id       ! File identifier
  INTEGER(HID_T) :: dset_id       ! Dataset identifier
  INTEGER(HID_T) :: dspace_id     ! Dataspace identifier

  INTEGER(HSIZE_T), DIMENSION(2) :: dims = (/4,6/) ! Dataset dimensions
  INTEGER     ::   rank = 2                        ! Dataset rank
  INTEGER     ::   error ! Error flag
  INTEGER     ::   i, j

  INTEGER, DIMENSION(4,6) :: dset_data, data_out ! Data buffers
  INTEGER(HSIZE_T), DIMENSION(2) :: data_dims
```

Fortran Code: h5_rdwt.f90 (cont'd)

```
! Initialize FORTRAN interface
CALL h5open_f(error)
! Create a new file using default properties
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error)
! Create the dataspace
CALL h5screate_simple_f(rank, dims, dspace_id, error)
! Create the dataset with default properties
CALL h5dcreate_f(file_id, dsetname, H5T_NATIVE_INTEGER, dspace_id, &
                 dset_id, error)  ! Initialize the dset_data array.
DO i = 1, 4
   DO j = 1, 6
      dset_data(i,j) = (i-1)*6 + j
   END DO
END DO

! Write the dataset
data_dims(1) = 4
data_dims(2) = 6
CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, dset_data, data_dims, error)
```

Fortran Code: h5_rdwt.f90 (cont'd)

```
   ! Read the dataset
   CALL h5dread_f(dset_id, H5T_NATIVE_INTEGER, data_out, data_dims, error)

   ! Close the dataset
   CALL h5dclose_f(dset_id, error)

   ! Terminate access to the dataspace
   CALL h5sclose_f(dspace_id, error)

   ! Close the file
   CALL h5fclose_f(file_id, error)

   ! Close FORTRAN interface
   CALL h5close_f(error)

END PROGRAM H5_RDWT
```

Try it out on Hyades

```
$ h5fc -o h5_rdwt.x h5_rdwt.f90
$ h5fc -show
gfortran -I/usr/include -L/usr/lib64 -lhdf5hl_fortran -lhdf5_hl -lhdf5_fortran \
         -lhdf5 -lrt -lsz -lz -ldl -lm -Wl,-rpath -Wl,/usr/lib64
```

or

```
$ gfortran -o h5_rdwt.x h5_rdwt.f90 -I/usr/include -lhdf5_fortran -lhdf5
$ ./h5_rdwt.x
$ h5dump dsetf.h5
HDF5 "dsetf.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 6, 4 ) / ( 6, 4 ) }
      DATA {
      (0,0): 1, 7, 13, 19,
      (1,0): 2, 8, 14, 20,
      (2,0): 3, 9, 15, 21,
      (3,0): 4, 10, 16, 22,
      (4,0): 5, 11, 17, 23,
      (5,0): 6, 12, 18, 24
      }}}}
```

C++ Code: h5_rdwt.cpp

```
#include <iostream>
#include <string>
#include "H5Cpp.h"
#ifndef H5_NO_NAMESPACE
    using namespace H5;
#endif

const H5std_string FILE_NAME("dset.h5");
const H5std_string DATASET_NAME("dset");
const int     NX = 4;                    // dataset dimensions
const int     NY = 6;
const int     RANK = 2;

int main (void) {
    // Data initialization
    int i, j;
    int data[NX][NY];          // buffer for data to write
    for (j = 0; j < NX; j++)
      for (i = 0; i < NY; i++)
       data[j][i] = i * 6 + j + 1;
```

C++ Code: h5_rdwt.cpp (cont'd)

```cpp
    // Try block to detect exceptions raised by any of the calls inside it
    try
    {
      // Turn off the auto-printing when failure occurs
      Exception::dontPrint();
      // Create a new file using the default property lists
      H5File file(FILE_NAME, H5F_ACC_TRUNC);
      // Create the data space for the dataset
      hsize_t dims[2];                 // dataset dimensions
      dims[0] = NX;
      dims[1] = NY;
      DataSpace dataspace(RANK, dims);
      // Create the dataset
      DataSet dataset = file.createDataSet(DATASET_NAME, PredType::STD_I32BE,
                                           dataspace);
      // Write the data to the dataset
      dataset.write(data, PredType::NATIVE_INT);
      // Read the dataset
      dataset.read(data, PredType::NATIVE_INT);
    }  // end of try block
```

C++ Code: h5_rdwt.cpp (cont'd)

```
    // catch failure caused by the H5File operations
    catch(FileIException error)
    {
       error.printError();
       return -1;
    }
    // catch failure caused by the DataSet operations
    catch(DataSetIException error)
    {
       error.printError();
       return -1;
    }
    // catch failure caused by the DataSpace operations
    catch(DataSpaceIException error)
    {
       error.printError();
       return -1;
    }
    return 0;  // successfully terminated
}
```

Try it out on Hyades

```
$ h5c++ -o h5_rdwt.x h5_rdwt.cpp
$ h5c++ -show
g++ -D_LARGEFILE64_SOURCE -D_LARGEFILE_SOURCE -L/usr/lib64 -lhdf5_hl_cpp \
    -lhdf5_cpp -lhdf5_hl -lhdf5 -lrt -lsz -lz -ldl -lm -Wl,-rpath -Wl,/usr/lib64
```

or

```
$ g++ -o h5_rdwt.x h5_rdwt.cpp -lhdf5_cpp -lhdf5
$ ./h5_rdwt.x
$ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 1, 7, 13, 19, 25, 31,
      (1,0): 2, 8, 14, 20, 26, 32,
      (2,0): 3, 9, 15, 21, 27, 33,
      (3,0): 4, 10, 16, 22, 28, 34
      }
   }
}}
```

Python Code: h5_rdwt.py

```
import h5py
import numpy as np

# Create a new file using defaut properties
file = h5py.File('dset.h5','w')

# Create a dataset under the Root group
dataset = file.create_dataset("dset",(4, 6), h5py.h5t.STD_I32BE)
print "Dataset dataspace is", dataset.shape
print "Dataset Numpy datatype is", dataset.dtype
print "Dataset name is", dataset.name
print "Dataset is a member of the group", dataset.parent
print "Dataset was created in the file", dataset.file

# Initialize data object with 0
data = np.zeros((4,6))
```

Python Code: h5_rdwt.py (cont'd)

```
# Assign new values
for i in range(4):
    for j in range(6):
        data[i][j]= i*6+j+1

# Write data
print "Writing data..."
dataset[...] = data

# Read data back and print it
print "Reading data back..."
data_read = dataset[...]
print "Printing data..."
print data_read

# Close the file before exiting
file.close()
```

Try it out on Hyades

```
$ module load python
$ python h5_rdwt.py
Dataset dataspace is (4, 6)
Dataset Numpy datatype is >i4
Dataset name is /dset
Dataset is a member of the group <HDF5 group "/" (1 members)>
Dataset was created in the file <HDF5 file "dset.h5" (mode r+)>
Writing data...
Reading data back...
Printing data...
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
```

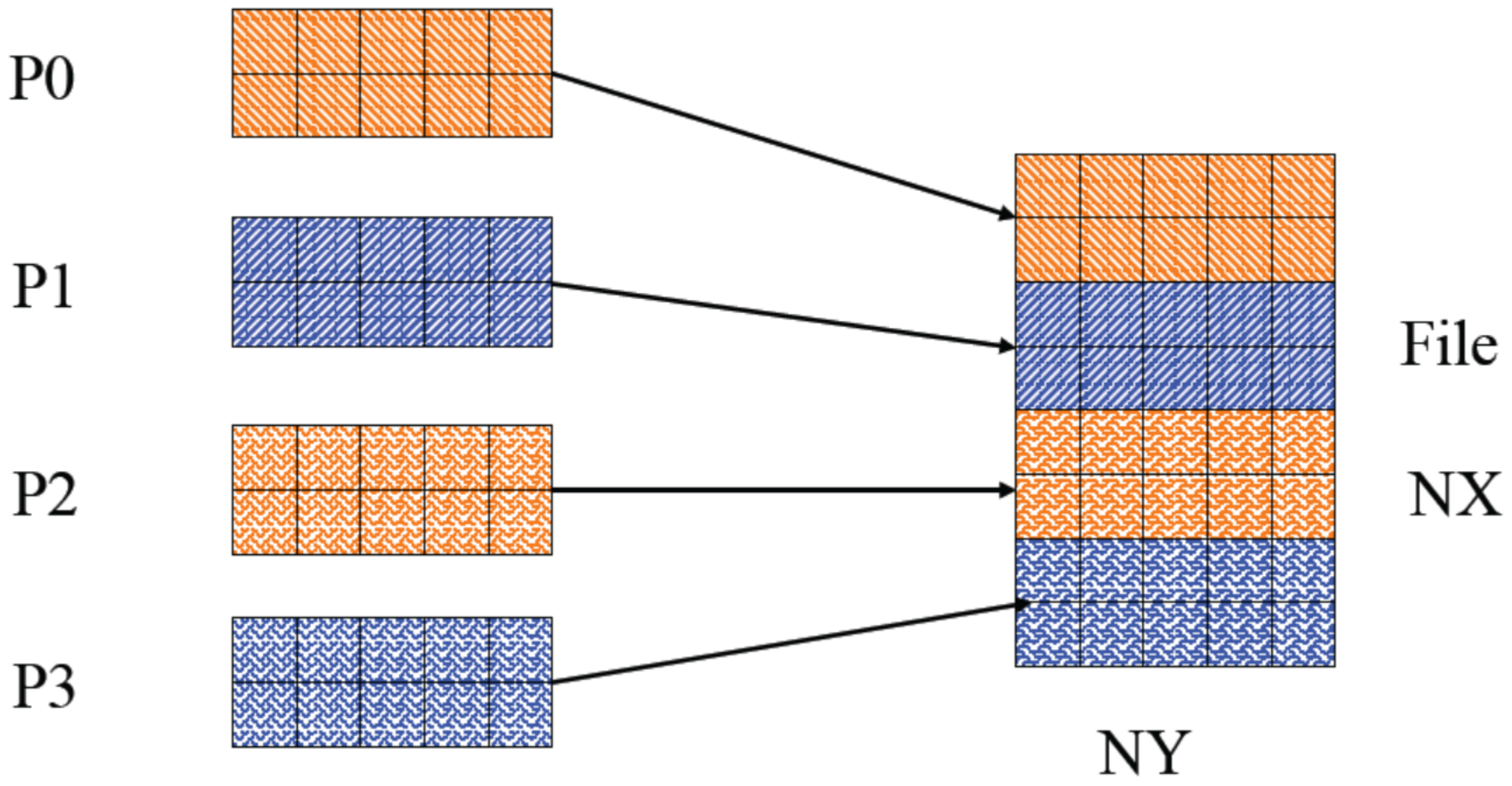
Parallel HDF5

- Parallel HDF5 (pHDF5) files are compatible with serial HDF5 files
  - sharable between different serial and parallel platforms
- A standard parallel interface portable to different platforms
- pHDF5 is built on top of MPI-IO, and can use MPI-IO optimizations
- A single file image to all processes, rather than having one file per process

Inside PHDF5

- **MPI_File_open** used to open file
- In **H5Dwrite**:
  - Processes communicate to determine file layout
    - Process 0 performs metadata updates
  - Call **MPI_File_set_view**
  - Call **MPI_File_write_all** to *collectively* write
- Memory hyperslab could be used to define noncontiguous region in memory
- At the MPI-IO layer:
  - Metadata updates at every write are a bit of a bottleneck
    - MPI-IO from process 0 introduces some skew
  - Use MPI-IO settings to tune the performance
    - Use **MPI_Info** object to control # of writes, # of stripes (Lustre), stripe size (Lustre), etc.

Example: Writing dataset by rows



C Code: write_grid_rows.c

```c
/*
 *   Writing out data. Run this example on 4 processor cores
 */
#include "mpi.h"
#include "hdf5.h"
#define FILENAME "grid_rows.h5"
#define NX 8
#define NX_L 4
#define NY 5
#define RANK 2
int main(int argc, char** argv) {
  hid_t       dset_id, memspace, filespace, file_id, plist_id;
  herr_t      status;
  int         i, j, my_proc, num_procs;
  double      grid_data[NX_L][NY];
  hsize_t     dimsf[RANK], offset[RANK], stride[RANK], count[RANK];

  /* initialize MPI */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

C Code: write_grid_rows.c (cont'd)

```c
/* create access property list */
plist_id = H5Pcreate(H5P_FILE_ACCESS);
/* necessary for parallel access */
status = H5Pset_fapl_mpio(plist_id, MPI_COMM_WORLD, MPI_INFO_NULL);
/* create an hdf5 file */
file_id = H5Fcreate(FILENAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
status = H5Pclose(plist_id);
/* initialize the grid */
for (i = 0; i < NX_L; i++)
  for (j = 0; j < NY; j++)
     grid_data[i][j] = 1.0*my_proc + 18;
/* create the dataspace */
dimsf[0] = NX;
dimsf[1] = NY;
filespace = H5Screate_simple(RANK, dimsf, NULL);
/* create a dataset */
dset_id = H5Dcreate(file_id, "dataset1", H5T_NATIVE_DOUBLE, filespace,
                    H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
```

C Code: write_grid_rows.c (cont'd)

```
plist_id = H5Pcreate(H5P_DATASET_XFER);
/* The other option is HDFD_MPIO_INDEPENDENT */
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

/* number of contiguous elements to write */
count[0] = dimsf[0]/num_procs;
count[1] = dimsf[1];
/* array which holds the starting position for each dimension */
offset[0] = my_proc * count [0];
offset[1] = 0;
/* array which holds the stride */
stride[0] = 1;
stride[1] = 1;

/* create the local memory space */
memspace = H5Screate_simple(RANK, count, NULL);
filespace = H5Dget_space(dset_id);
```

C Code: write_grid_rows.c (cont'd)

```
  /* create the hyperslab -- says how you want to lay out data */
  status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset,
                               NULL, count, NULL);

  status = H5Dwrite(dset_id, H5T_NATIVE_DOUBLE, memspace,
                    filespace, H5P_DEFAULT, grid_data);

  status = H5Dclose(dset_id);

  H5Sclose(filespace);
  H5Sclose(memspace);

  status = H5Fclose(file_id);

  MPI_Finalize();
}
```

More pHDF5 examples on NERSC systems at: /project/projectdirs/training/NUG2010/pHDF5_examples.tar

Try them out on Cori Phase I

```
cori03:> cc readFile1.c -o readFile1.x
cori03:> ftn readFile2.f90 -o readFile2.x
cori03:> cc writeFile1.c -o writeFile1.x
cori03:> ftn writeFile2.f90 -o writeFile2.x

cori03:> salloc -N 1 -p debug -L SCRATCH -C haswell

nid00468:> srun -n 2 ./writeFile1.x
nid00468:> srun -n 4 ./readFile2.x
nid00468:> rm datafile
nid00468:> srun -n 4 ./writeFile2.x
nid00468:> srun -n 2 ./readFile1.x
```

Try it out on Cori Phase I

```
cori03:> module load cray-hdf5-parallel

cori03:> cc -o write_grid_rows.x write_grid_rows.c

cori03:> salloc -N 1 -p debug -L SCRATCH -C haswell

nid00468:> srun -n 4 ./write_grid_rows.x
```

http://www.nersc.gov/users/data-analytics/data-management/i-o-libraries/hdf5-2/hdf5/

Try it out on Hyades

```
$ module load hdf5/p_impi_intel_1.8.10.1

$ mpiicc -o write_grid_rows.x write_grid_rows.c -lhdf5 -lz

$ mpirun -n 4 ./write_grid_rows.x

$ h5dump grid_rows.h5
```

Fortran Code: write_grid_rows.f90

```fortran
PROGRAM WRITE_GRID_ROWS_F
  Use HDF5
  implicit none
  include 'mpif.h'

  character(len=25), PARAMETER :: filename = "grid_rows_f.h5"
  ! Note the X and Y are reversed from the C program
  integer, PARAMETER :: NX = 5
  integer, PARAMETER :: NY = 8
  integer(HID_T) :: file_id, plist_id, dset_id, memspace, filespace
  integer(HSIZE_T), DIMENSION(2) :: dimsf, offset, stride, count
  integer, allocatable :: grid_data(:,:)
  integer :: rank, i, j, status, error
  integer :: num_procs, my_proc

  ! initialize MPI
  call mpi_init(error)
  call mpi_comm_size(MPI_COMM_WORLD, num_procs, error)
  call mpi_comm_rank(MPI_COMM_WORLD, my_proc, error)
```

Fortran Code: write_grid_rows.f90 (cont’d)

```fortran
! initialize Fortran interface
call h5open_f(error)
! setup file access property list for MPI-IO access
call h5pcreate_f(H5P_FILE_ACCESS_F, plist_id, error)
call h5pset_fapl_mpio_f(plist_id, MPI_COMM_WORLD, MPI_INFO_NULL, error)
! create the file collectively
call h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error, access_prp=plist_id)

dimsf(1) = NX;
dimsf(2) = NY;
allocate(grid_data(NX,NY/num_procs))
do i=1, NX
   do j =1, NY/num_procs
      grid_data(i,j) = my_proc
   end do
end do

rank = 2
! create the file space
call h5screate_simple_f(rank, dimsf, filespace, error)
```

Fortran Code: write_grid_rows.f90 (cont'd)

```
! create the dataset with default properties
call h5dcreate_f(file_id, "dataset1", H5T_NATIVE_INTEGER, filespace, dset_id, &
                 error)
! create property list for collective dataset write
call h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
call h5pset_dxpl_mpio_f(plist_id, H5FD_MPIO_COLLECTIVE_F, error)

count(1) = dimsf(1)
count(2) = dimsf(2)/num_procs
offset(1) = 0
offset(2) = my_proc * count(2)
! create the memory space
call h5screate_simple_f(rank, count, memspace, error)
call h5dget_space_f(dset_id, filespace, error)

! create the hyperslab
call h5sselect_hyperslab_f(filespace, H5S_SELECT_SET_F, offset, count, error)
call h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, grid_data, count, error, &
       file_space_id = filespace, mem_space_id = memspace, xfer_prp=plist_id)
```

Fortran Code: write_grid_rows.f90 (cont'd)

```fortran
    ! close resources
    call h5sclose_f(filespace, error)
    call h5sclose_f(memspace, error)
    call h5dclose_f(dset_id, error)
    call h5pclose_f(plist_id, error)

    ! close the file
    call h5fclose_f(file_id, error)

    ! close Fortran Interface
    call h5close_f(error)

    call MPI_FINALIZE(error)
end PROGRAM WRITE_GRID_ROWS_F
```

Try it out on Cori Phase I

```
cori03:> module load cray-hdf5-parallel

cori03:> ftn -o write_grid_rows.x write_grid_rows.f90

cori03:> salloc -N 1 -p debug -L SCRATCH -C haswell

nid00468:> srun -n 4 ./write_grid_rows.x
```

http://www.nersc.gov/users/data-analytics/data-management/i-o-libraries/hdf5-2/hdf5/

Try it out on Hyades

```
$ module load hdf5/p_impi_intel_1.8.10.1

$ mpiifort -o write_grid_rows.x write_grid_rows.f90 \
         -lhdf5_fortran -lhdf5 -lz

$ mpirun -n 4 ./write_grid_rows.x

$ h5dump grid_rows.h5
```

Other High-Level I/O Libraries

- Parallel netCDF (PnetCDF)
  - http://cucis.ece.northwestern.edu/projects/PnetCDF/
  - A parallel I/O library for accessing NetCDF files in CDF, CDF-2 and CDF-5 formats
  - Tutorial: http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial
- NetCDF-4
  - http://www.unidata.ucar.edu/software/netcdf/
  - netCDF API with HDF5 back-end
  - Parallel I/O: http://www.unidata.ucar.edu/software/netcdf/docs/parallel_io.html
- ADIOS (Adaptable IO System)
  - https://www.olcf.ornl.gov/center-projects/adios/
  - Configurable (XML) I/O approaches
  - Tutorial: https://www.nersc.gov/assets/Uploads/W09-norbert-oakland-userforum-2014.pdf
- Silo
  - https://wci.llnl.gov/simulation/computer-codes/silo
  - A mesh and field I/O library and scientific database
- etc.

Further Readings

- **Parallel I/O for High Performance Computing**, *by* John M. May, Morgan Kaufmann Publishers, 2001
- **High Performance Parallel I/O**, *edited by* Prabhat & Quincey Koziol, Chapman and Hall/CRC, 2014
- *Chapter 7 (Parallel I/O)* of **Using Advanced MPI: Modern Features of the Message-Passing Interface**, *by* Gropp, Hoefler, Thakur, and Lusk, MIT Press, 2014.
  http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981848
- **Parallel I/O in Practice**: https://www.nersc.gov/assets/Training/pio-in-practice-sc12.pdf
- **ROMIO**: http://www.mcs.anl.gov/projects/romio/
- **Intro to HDF5**, *by* Katie Antypas (NERSC):
  https://www.nersc.gov/assets/NUG-Meetings/HDF5-tutorialNUG2010.pdf
- **Introduction to HDF5**: https://www.hdfgroup.org/HDF5/doc/H5.intro.html
- **HDF5 Software Documentation**: https://www.hdfgroup.org/HDF5/doc/index.html
- **Parallel HDF5**: https://www.hdfgroup.org/HDF5/PHDF5/
- **Burst Buffer Tutorials and Example Batch Scripts:**
  http://www.nersc.gov/users/computational-systems/cori/burst-buffer/example-batch-scripts/