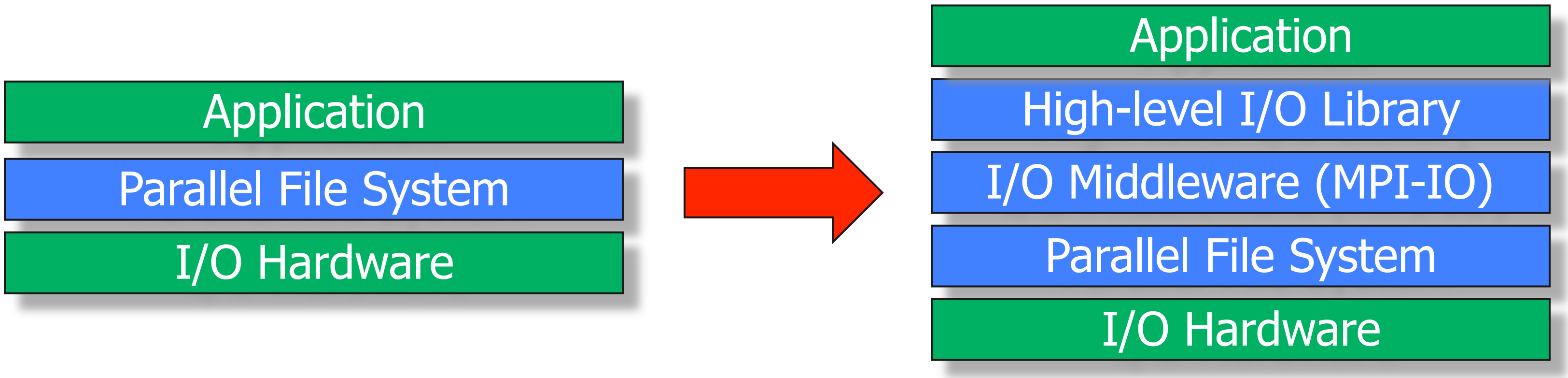
Parallel I/O and MPI-IO

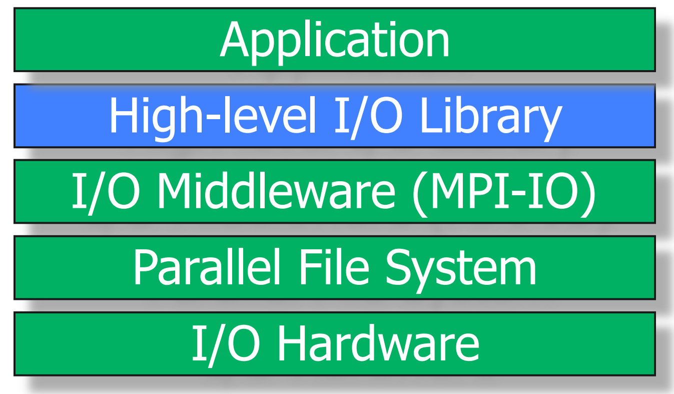
Rajeev Thakur

I/O for Computational Science



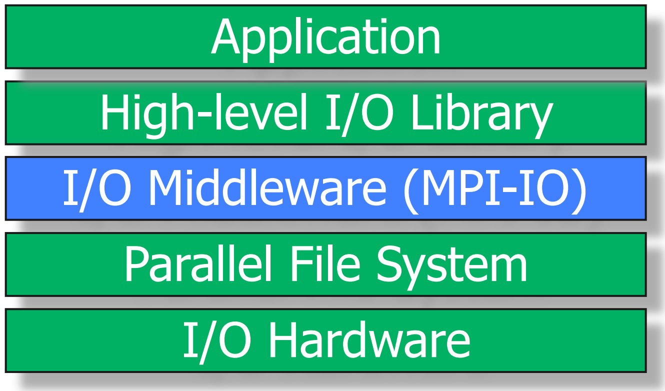
- Break up support into multiple layers with distinct roles:
  - High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)
  - Middleware layer deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
  - Parallel file system maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)

High Level Libraries



- Examples: HDF-5, PnetCDF
- Provide an appropriate abstraction for domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Self-describing, structured file format
- Map to middleware interface
  - Encourage collective I/O
- Provide optimizations that middleware cannot
  - e.g. caching attributes of variables

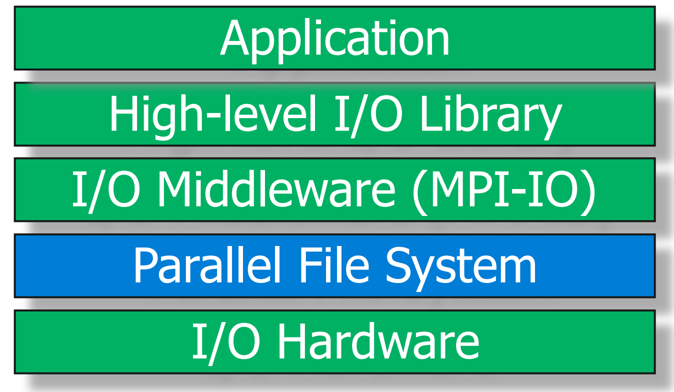
I/O Middleware



- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Match the underlying programming model (e.g. MPI)
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs

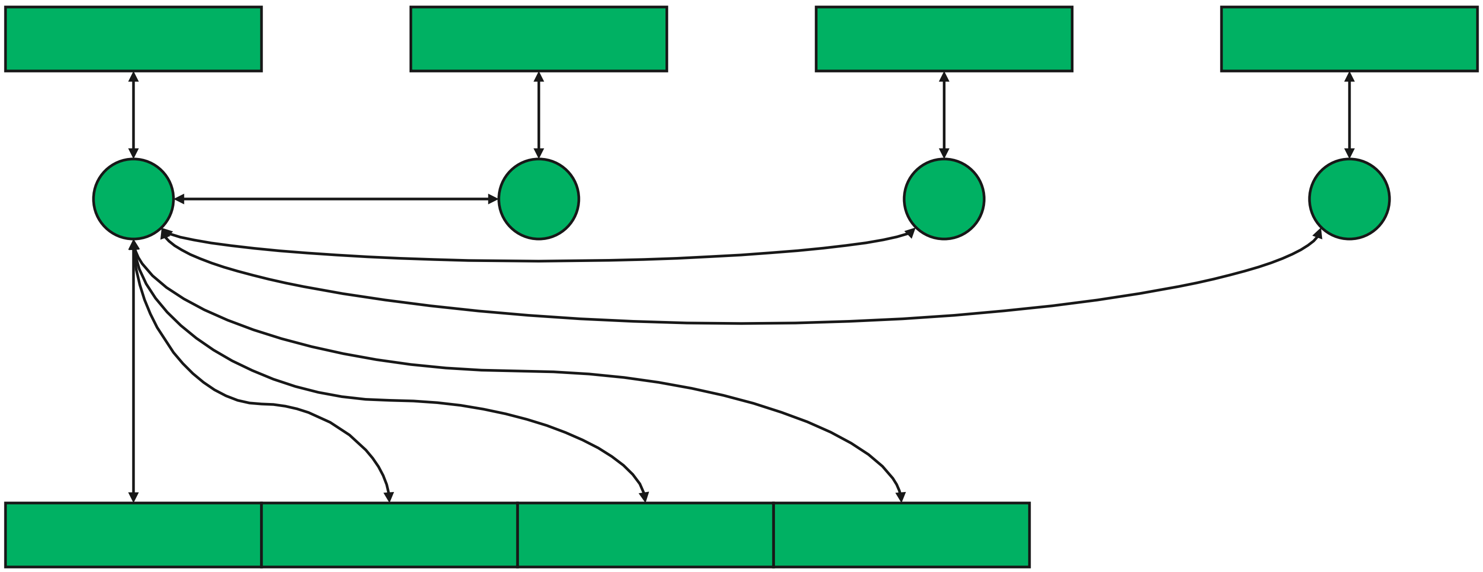
Parallel File System

- Manage storage hardware
  - Present single view
  - Focus on concurrent, independent access
  - Knowledge of collective I/O usually very limited



- In the context of computational science, publish an interface that middleware can use effectively
  - Rich I/O language
  - Relaxed but sufficient semantics

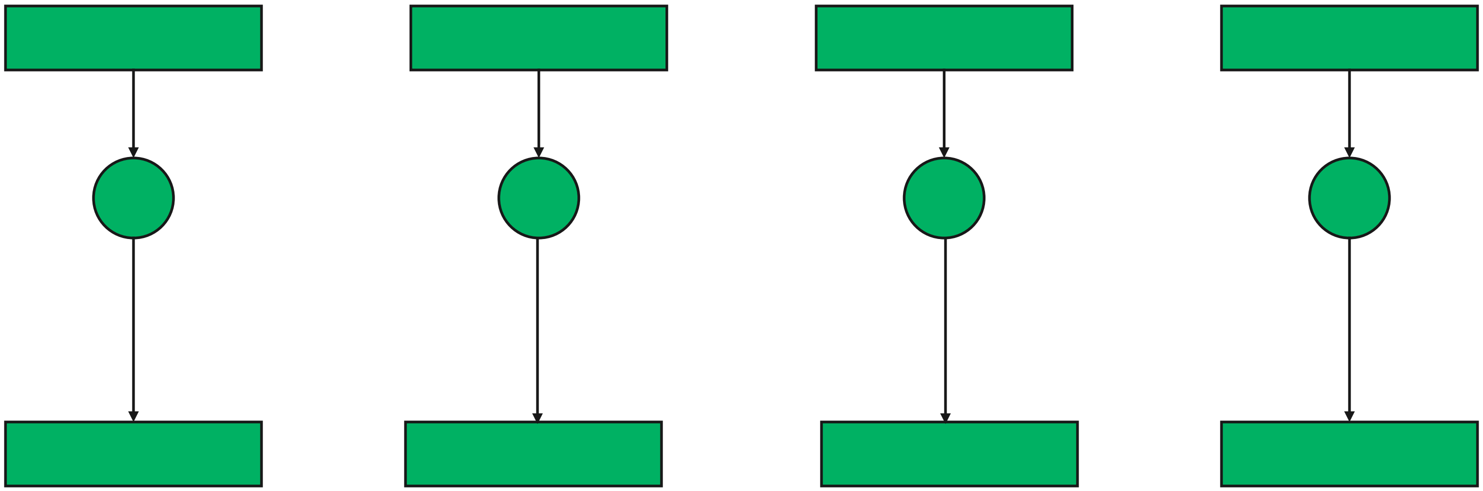
Non-Parallel I/O



- Non-parallel
- Pro: Results in a single file
- Con: Poor performance
- Legacy from before application was parallelized

Independent Parallel I/O

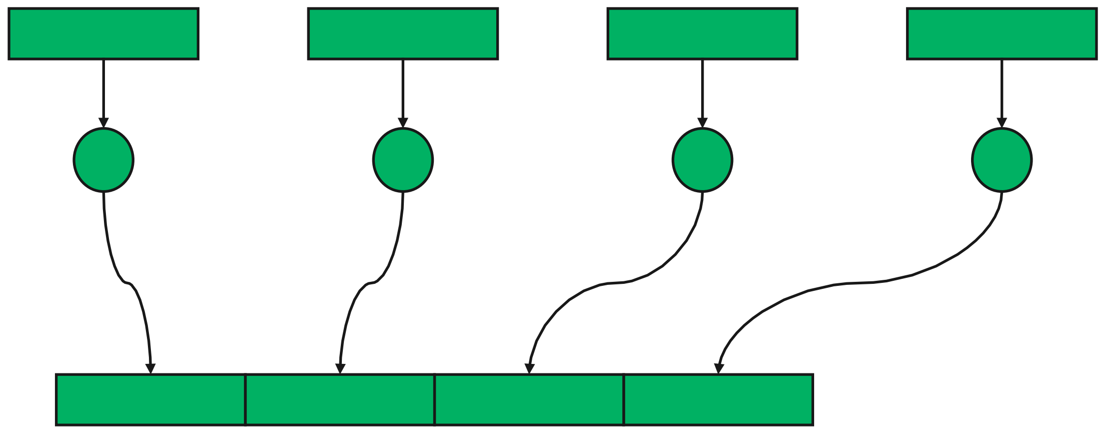
- Each process writes to a separate file



  - Pro: parallelism
  - Con: lots of small files to manage
  - Legacy from before MPI-IO

What is Parallel I/O?

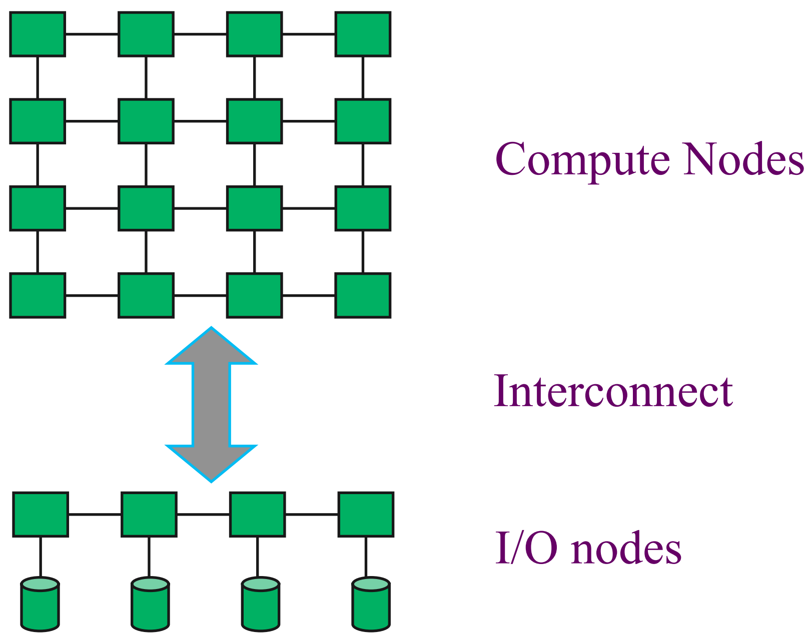
- From user's perspective:
  - Multiple processes or threads of a parallel program accessing data concurrently from a *common* file



- Results in a single file and you can get good performance

What is Parallel I/O?

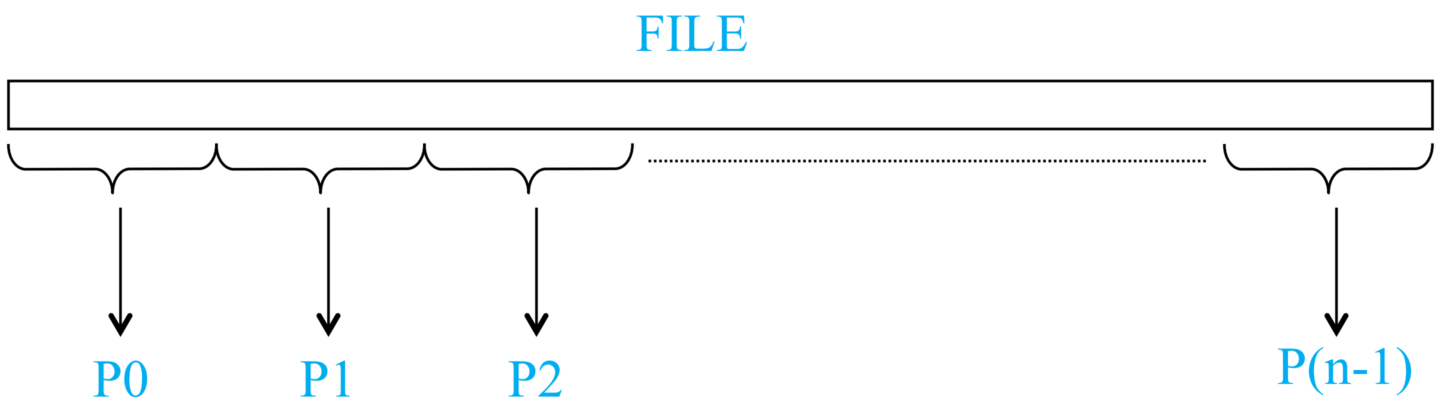
- From system perspective:
  - Files striped across multiple I/O servers
  - File system designed to perform well for concurrent writes and reads (parallel file system)



Why MPI is a Good Setting for Parallel I/O

- Writing is like sending and reading is like receiving
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- i.e., lots of MPI-like machinery

Using MPI for Simple I/O



Each process needs to read a chunk of data from a common file

Using Individual File Pointers

```
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);
```

Using Explicit Offsets

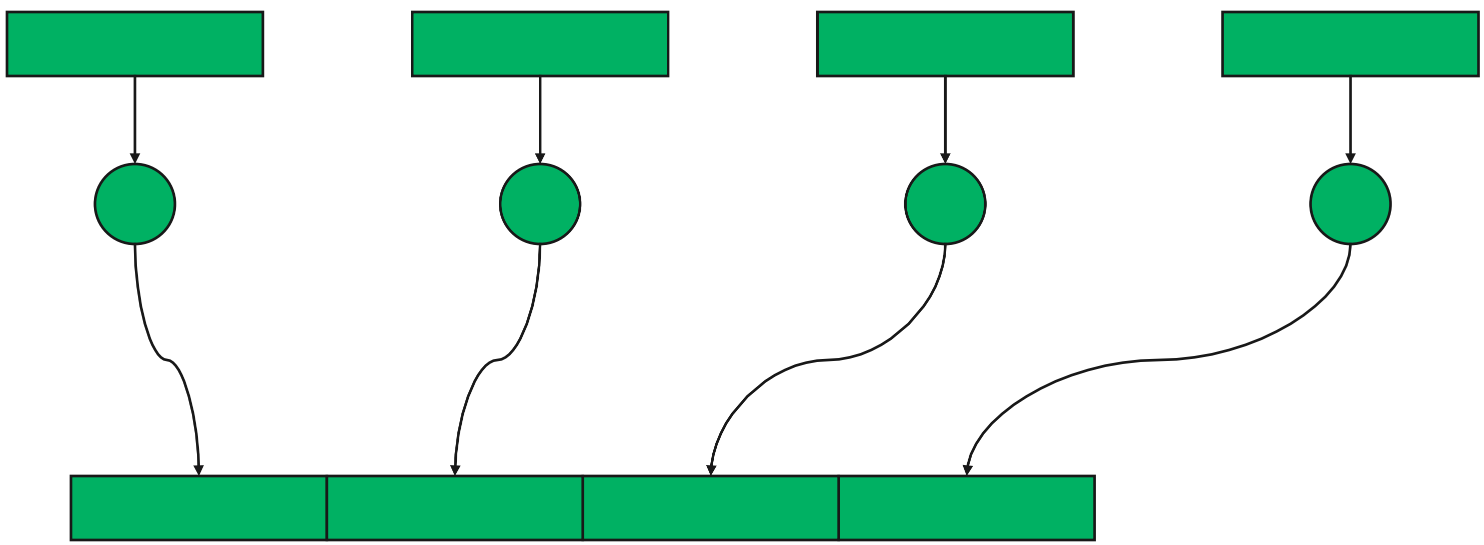
```
include 'mpif.h'

integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset
C in F77, see implementation notes (might be integer*8)

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
          MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE
call MPI_FILE_READ_AT(fh, offset, buf, nints,
                    MPI_INTEGER, status, ierr)
call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count, 'integers'

call MPI_FILE_CLOSE(fh, ierr)
```

Writing to a File

- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- Use **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** as the flags to **`MPI_File_open`**
- If the file doesn't exist previously, the flag **`MPI_MODE_CREATE`** must also be passed to **`MPI_File_open`**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

Using File Views

- Processes write to shared file



- **`MPI_File_set_view`** assigns regions of the file to separate processes

File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

File View Example

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                  MPI_INT, MPI_INT, "native",
                 MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

MPI_File_set_view

- Describes that part of the file accessed by a single MPI process.
- Arguments to **`MPI_File_set_view`**:
  - **`MPI_File file`**
  - **`MPI_Offset disp`**
  - **`MPI_Datatype etype`**
  - **`MPI_Datatype filetype`**
  - **`char *datarep`**
  - **`MPI_Info info`**

Fortran Version

```
PROGRAM main

use mpi

integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
do i = 0, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo


* in F77, see implementation notes (might be integer*8)
```

Fortran Version contd.

```
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
                   MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                   MPI_INFO_NULL, thefile, ierr)
call MPI_TYPE_SIZE(MPI_INTEGER, intsize)
disp = myrank * BUFSIZE * intsize
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
                       MPI_INTEGER, 'native', &
                       MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, &
                       MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)

END PROGRAM main
```

Other Ways to Write to a Shared File

- **`MPI_File_seek`**
- **`MPI_File_read_at`**
- **`MPI_File_write_at`**
- **`MPI_File_read_shared`**
- **`MPI_File_write_shared`**

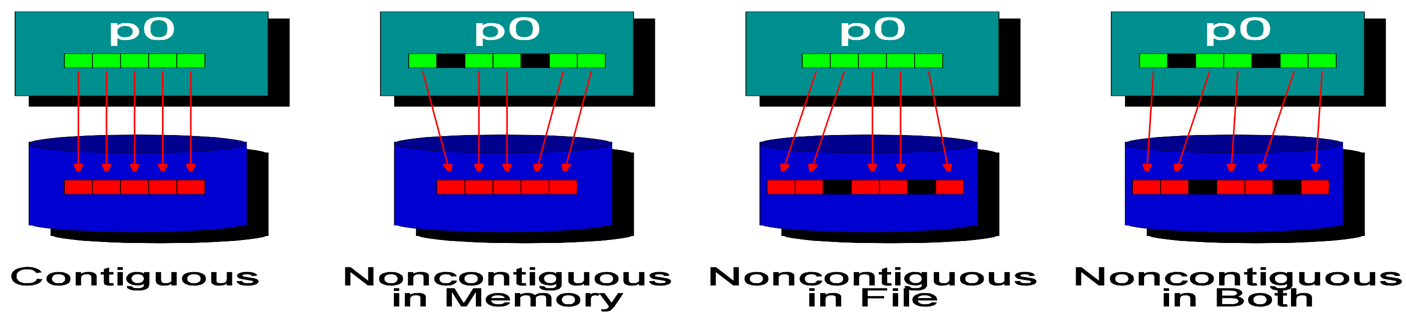
like Unix seek

combine seek and I/O
for thread safety

use shared file pointer

- Collective operations

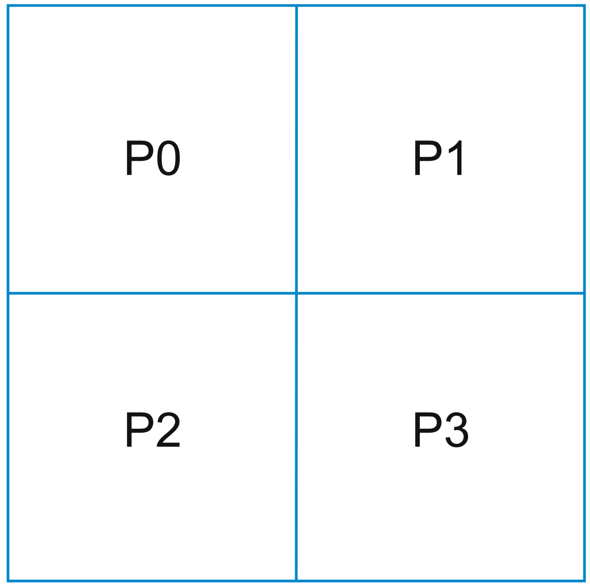
Noncontiguous I/O



- Contiguous I/O moves data from a single block in memory into a single region of storage
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O

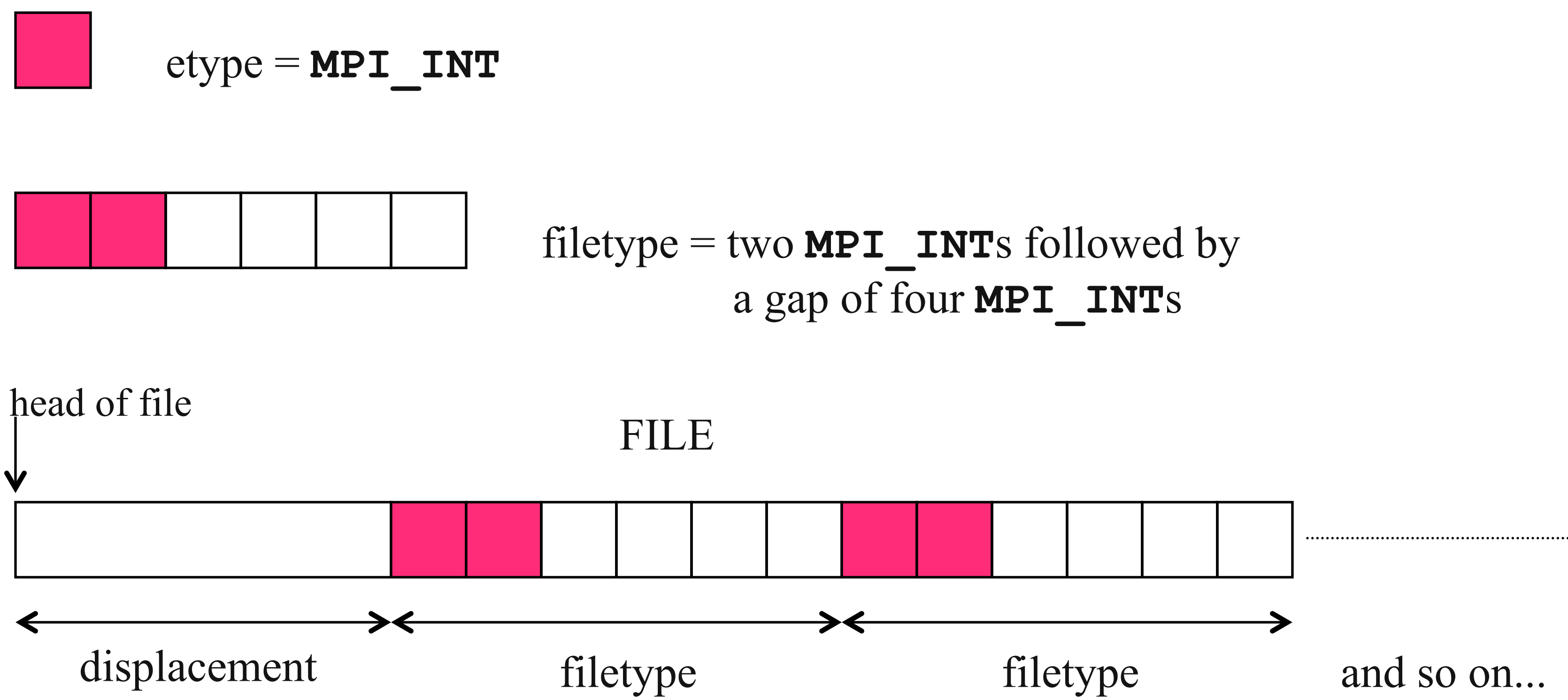
Example: Distributed Array Access

2D array distributed among four processes



File containing the global array in row-major order

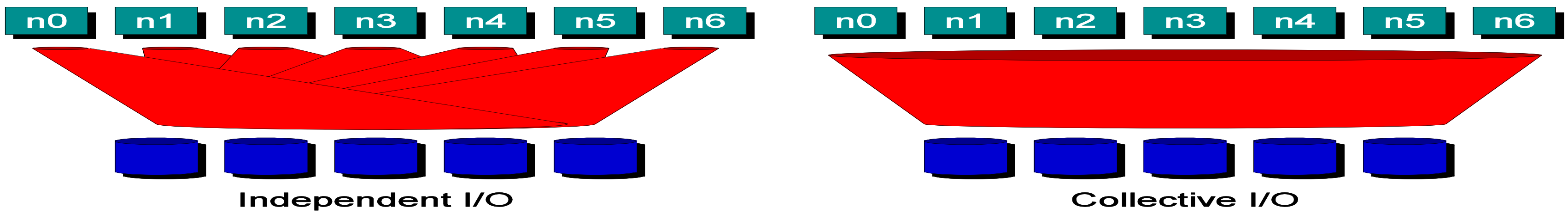
A Simple Noncontiguous File View Example



File View Code

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

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, "/pfs/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);
```

Collective I/O (1)



- Many applications have phases of computation and I/O
- During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions must be called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole
- Independent I/O is not organized in this way
  - No apparent order or structure to accesses

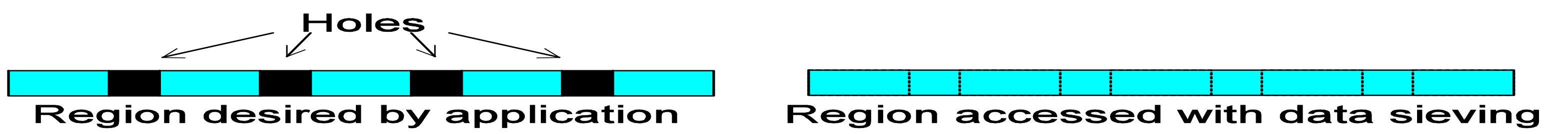
Collective I/O (2)

- **`MPI_File_read_all`**, **`MPI_File_read_at_all`**, etc
- **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
- Each process specifies only its own access information -- the argument list is the same as for the non-collective functions

Under the Covers of MPI-IO

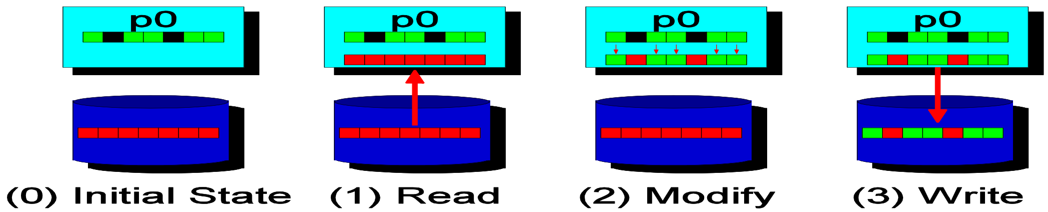
- MPI-IO implementation is given a lot of information in this case:
  - Collection of processes reading data
  - Structured description of the regions
- Implementation has some options for how to obtain this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

Data Sieving



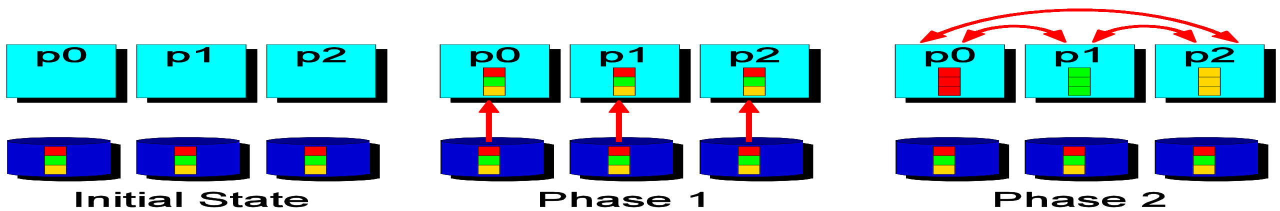
- Data sieving is used to combine lots of small accesses into a single larger one
  - Remote file systems (parallel or not) tend to have high latencies
  - Reducing # of operations important
- Generally very effective, but not as good as having a PFS that supports noncontiguous access

Data Sieving Writes



- Using data sieving for writes is more complicated
  - Must read the entire region first
  - Then make our changes
  - Then write the block back
- Requires locking in the file system
  - Can result in false sharing (interleaved access)
  - PFS supporting noncontiguous writes is preferred

Two-Phase Collective I/O

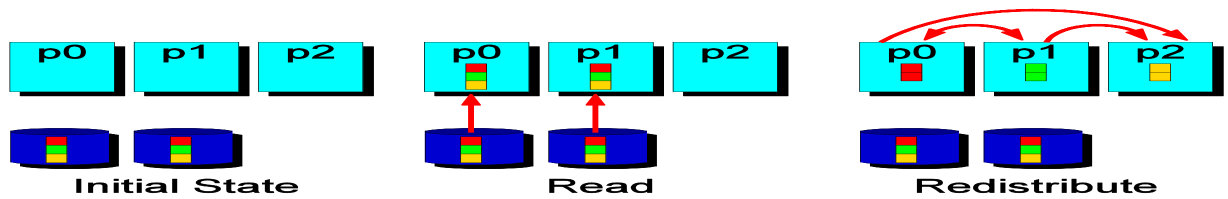


- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- Second ``phase'' moves data to final destinations

Two-Phase Writes

- Similarly to data sieving, we need to perform a read/modify/write for two-phase writes if *combined* data is noncontiguous
- Overhead is substantially lower than independent access to the same regions because there is little or no false sharing
- Note that two-phase is usually applied to file regions, not to actual blocks

Aggregation



- Aggregation refers to the more general application of this concept of moving data through intermediate nodes
  - Different #s of nodes performing I/O
  - Could also be applied to independent I/O
- Can also be used for remote I/O, where aggregator processes are on an entirely different system