Parallel Computing for Science & Engineering
Introduction to MPI, Parallel I/O
Spring 2018

Instructors:
Charlie Dey TACC
Lars Koesterke, TACC

Binary Data Files

- They have a name (e.g. myfile, data, results) in the filesystem.
- In a program, a filehandle (in C) or unit number (in Fortran) is used as a “file descriptor” for referencing the file.

```
open(fh/unit<name>)
```

- In a program seek (C) and position (Fortran) will position the file for reading or writing.
- When reading a file, the file data must agree with the declaration type of the variable, array, or structure representation which was used when writing them to disk.

Binary Data Files

```
#include<stdio.h>

int main(){
   int i,a[10];
   FILE *fh;

   for(i=0;i<10;i++) a[i]=i;

   fh = fopen("myfile","w");

 //fwrite(a,sizeof(int),10,fh);   //Can do this either way
   fwrite(a,sizeof(a),   1,fh);

   fclose(fh);
```

Binary Data Files

```
//  read first 2 int of file and put in into a[0] an a[1]

   fh = fopen("myfile","r");

   fread(&a[8],sizeof(int),1,fh);     // read position 1 in
                                      // file

   fread(&a[9],sizeof(int),1,fh);     // read position 2 in
                                      // file
   fclose(fh);
   for(i=0;i<10;i++) printf("%4d",a[i]); printf("\n");
```

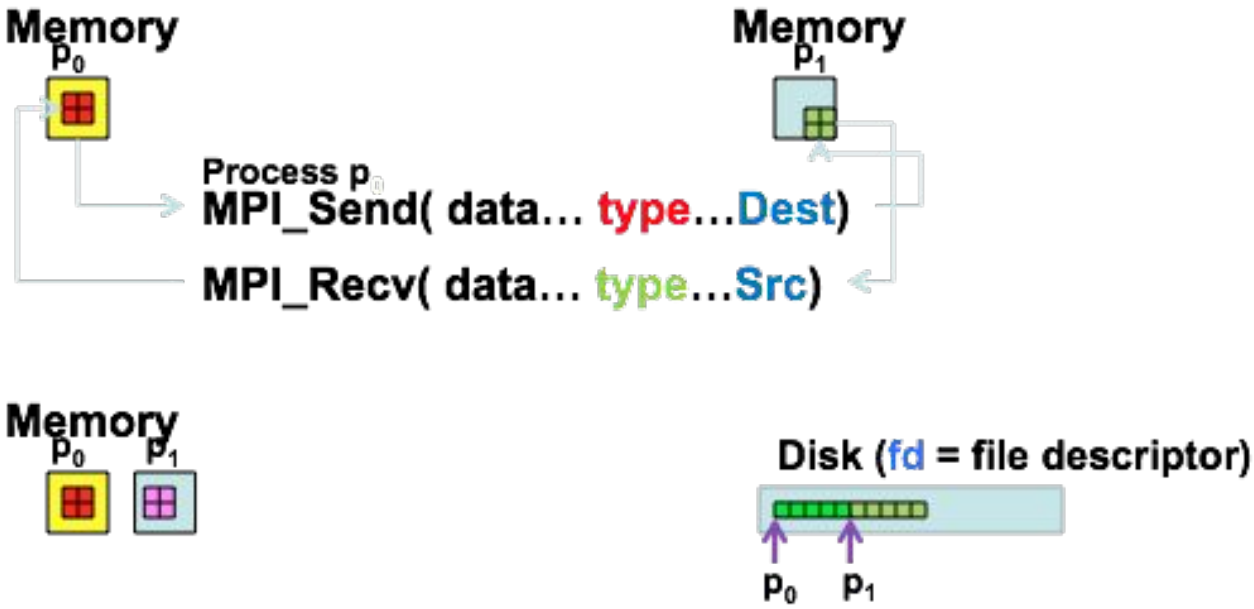
Binary Data Files

```
//  read last 2 ints of file and put into a[0] an a[1]

   for(i=0;i<10;i++) a[i]=i;
   fh = fopen("myfile","r");

   fseek(fh,sizeof(int)*8, SEEK_SET); // go to position 8
                                      // in file
   fread(&a[0],sizeof(int),1,fh);     // read, increment
                                      // pointer
    fread(&a[1],sizeof(int),1,fh);      // read from
                                     // position 9
   fclose(fh);
   for(i=0;i<10;i++) printf("%4d",a[i]); printf("\n");
}
```

MPI-IO General Concepts



File Systems

- Three general file system options:
  - Local
  - Remote (e.g. NFS)
  - Parallel (e.g. PVFS, LUSTRE, GPFS)
- MPI-IO - I/O part of MPI-2 Standard
- ROMIO - Implementation of MPI-IO
  - Handles mapping MPI-IO calls into communication (MPI) and file I/O

Local System

- Local - Use storage on each node's disk
  - Each node has own separate (disk) Unix file system
- Local - Access
  - Accessed only by application on node through OS
  - Relatively high performance
  - Files must be copied to/from each node
  - No synchronization
  - Most useful for temp/scratch files during job execution.
  - Not accessible after job.

Remote File Systems

- Remote - Use storage on server
  - Each node uses (disk) file system on a server
- Remote - Access
  - Every node sees same file system
  - Relatively low performance
  - Synchronization mechanisms manage changes
  - "Traditional" UNIX approach (NFS, AFS, etc.)
  - Doesn't scale well; server becomes bottleneck
  - Solution for small clusters (<100 nodes), reading/writing small files
  - Uses RAID devices for fault tolerance

Remote File Systems

- Remote - Use storage on server
  - Each node uses (disk) file system on a server
- Remote - Access
  - Every node sees same file system
  - Relatively low performance
  - Synchronization mechanisms manage changes
  - "Traditional" UNIX approach (NFS, AFS, etc.)
  - Doesn't scale well; server becomes bottleneck
  - Solution for small clusters (<100 nodes), reading/writing small files
  - Uses RAID devices for fault tolerance

Parallel File Systems

- Parallel - Use storage on servers
  - Each node uses (disks) file system on multiple servers (A single file system is mounted on multiple servers)
- Parallel - Access
  - Every node sees the same file system
  - Relatively high performance
  - Works best for I/O intensive applications
  - Not a good solution for small files
  - Servers may look identical to compute nodes; but usually have a “shared disk” file system (SAN, RAID array, etc.) to provide block level access from multiple servers.

Traditional I/O in a Parallel Environment

- Traditional I/O
  - lots of files
    - every process writes its own file
    - serial post-processing to combine/unify
  - one process, one file
    - everybody sends to 0
    - 0 collects, organizes and writes to a single file
  - both involve a completely serial step
  - both may fail to take advantage of the underlying file system

Parallel I/O Environment

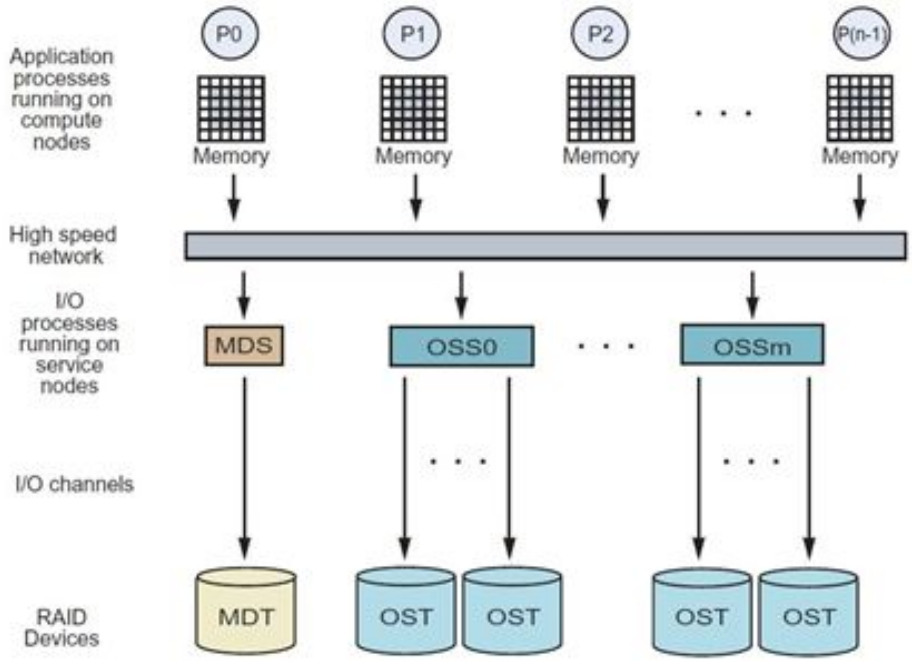
- Parallel I/O
  - everybody writes to a single file
    - writes only its portion
    - all go simultaneously
  - MPI I/O
    - portable
    - consistent
    - may be tuned for local features
  - parallel file systems
    - multiple servers (multiple disks)
    - automatic file striping
    - 10's of thousands of nodes, 10-100s of Petabytes
    - Top systems have TB/sec performance

Parallel File Systems(FS): Lustre

- Linux Cluster FS
  - 60 of Top 100 computers use it
  - open source
  - pay for support
- Linux-only
  - sits atop ext3
  - integrated into the kernel
- Supports striping & large blocks

Parallel File Systems: Lustre

- Object Storage Servers(OSS)
- Object Storage Targets (OST)
- MetaData Server (MDS)

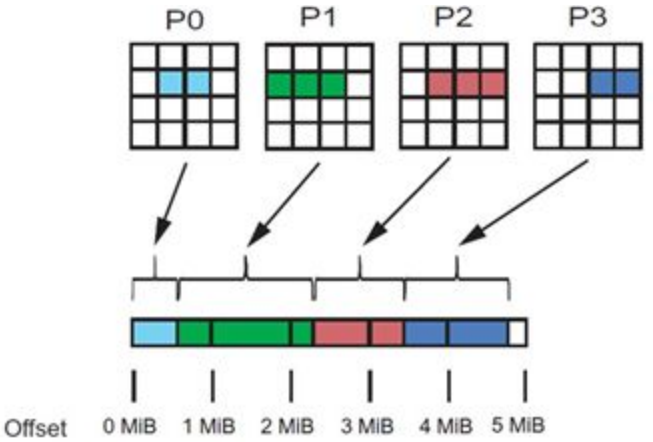


www.nics.tennessee.edu/io-tips

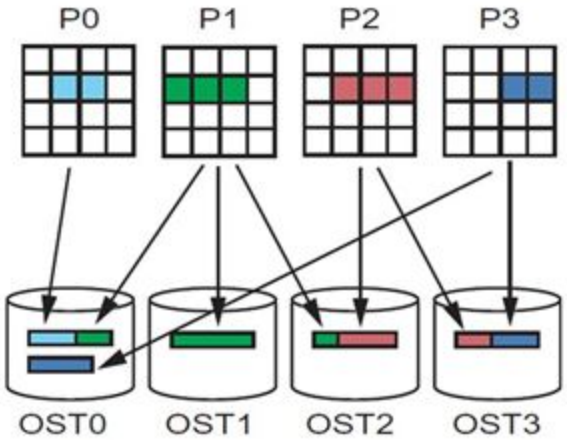
- Each File is stored in chunks across OSTs
- A MDS holds file storage info (metadata for open/close).
- Read/Write operations go directly to OST on a OSS

Parallel File Systems: Lustre

TACC Defaults:
Stripe size = 1MB
1-way striping on $WORK
4-way striping on $SCRATCH



Well-defined IO
uses "chunks"
equal to the Stripe
Size (and non-overlapping).



www.nics.tennessee.edu/io-tips

MPI-I/O

- Think of disks operations as slow message passing
  - Reads are receives and writes are sends.

- A parallel I/O system must have a mechanism to
  - define data layout in memory and file:
    - *MPI datatypes*
  - handle nonblocking operations:
    - *MPI wait (request) objects*
  - provide collective operations:
    - *MPI communicator context*

MPI-I/O Features

- Parallel Read/Write
- Blocking and Non-Blocking
- Non-Contiguous Access of Disk/Memory Data
- Collective Reads/Writes (across nodes)
- Portable Data Representation

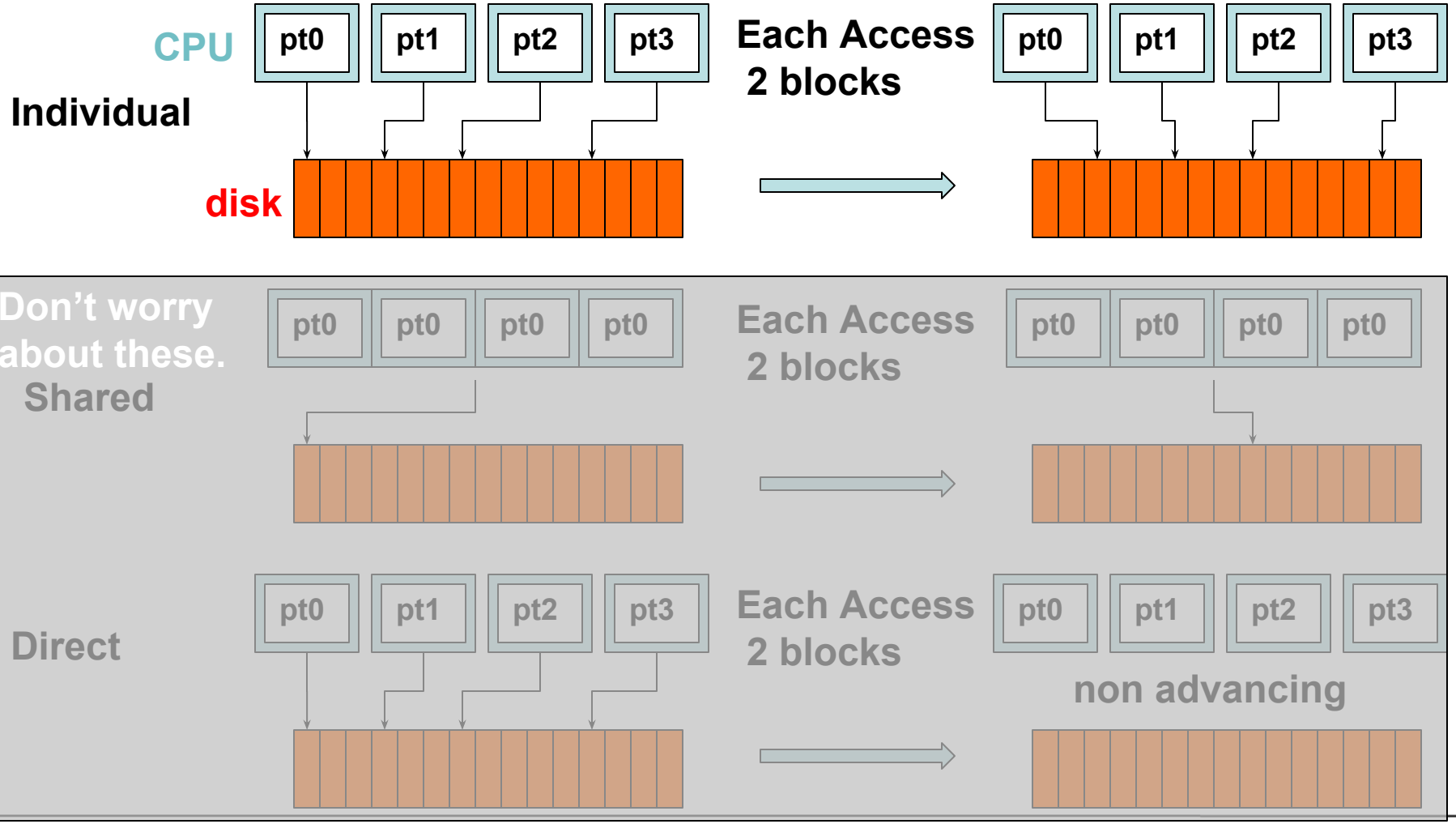
MPI-I/O Features

- Write/Read large blocks simultaneously.
- Use MPI derived data types for non-contiguous data (eliminates packing buffers).
- Non-blocking calls (overlap calculations with I/O)
- Use collective I/O calls (forms large blocks).
- Provide hints with “info” parameter (HW options).
- Provide a consistent view of the I/O operation for all processes.

MPI-IO

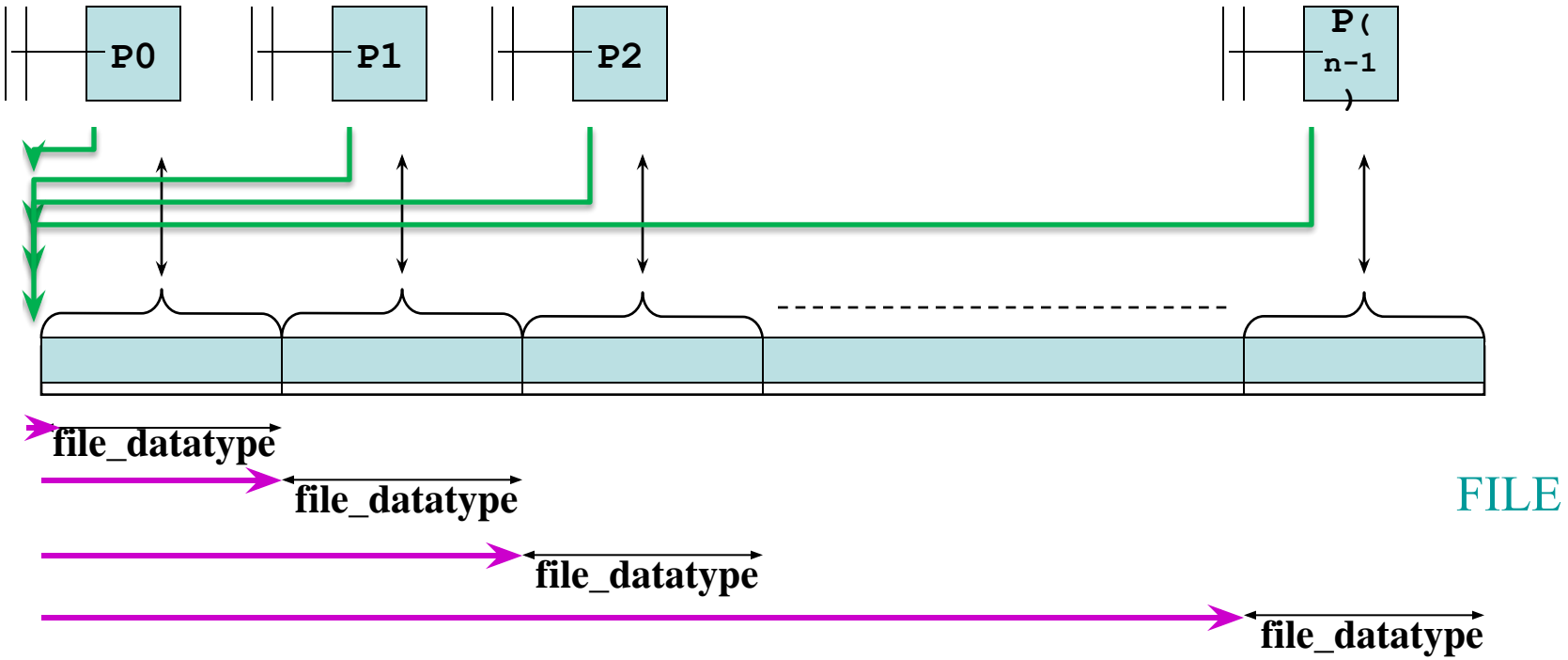
- Read/Write simultaneously, uses File Position
  - Individual (advancing, uses seek)
  - Shared (advancing, synchronized)
  - Direct Offset (set in read/write statement)
- Views (partitioning is expressed by datatypes for file file_datatype)
  - Each process has its own view (partition)
  - Positioning Data By Fileview Displacement
- Initialization (associates file_datatype & info with filehandle)
- Flushing Data
- Collective File I/O

MPI-IO File


CPU
Individual
pt0
pt1
pt2
pt3
disk
Each Access
2 blocks
pt0
pt1
pt2
pt3
Don't worry
about these.
Shared
pt0
pt0
pt0
pt0
Each Access
2 blocks
pt0
pt0
pt0
pt0
Direct
pt0
pt1
pt2
pt3
Each Access
2 blocks
pt0
pt1
pt2
pt3
non advancing

Blocked

Simple Approach: or Fileview Displacement (default pointers at 0)



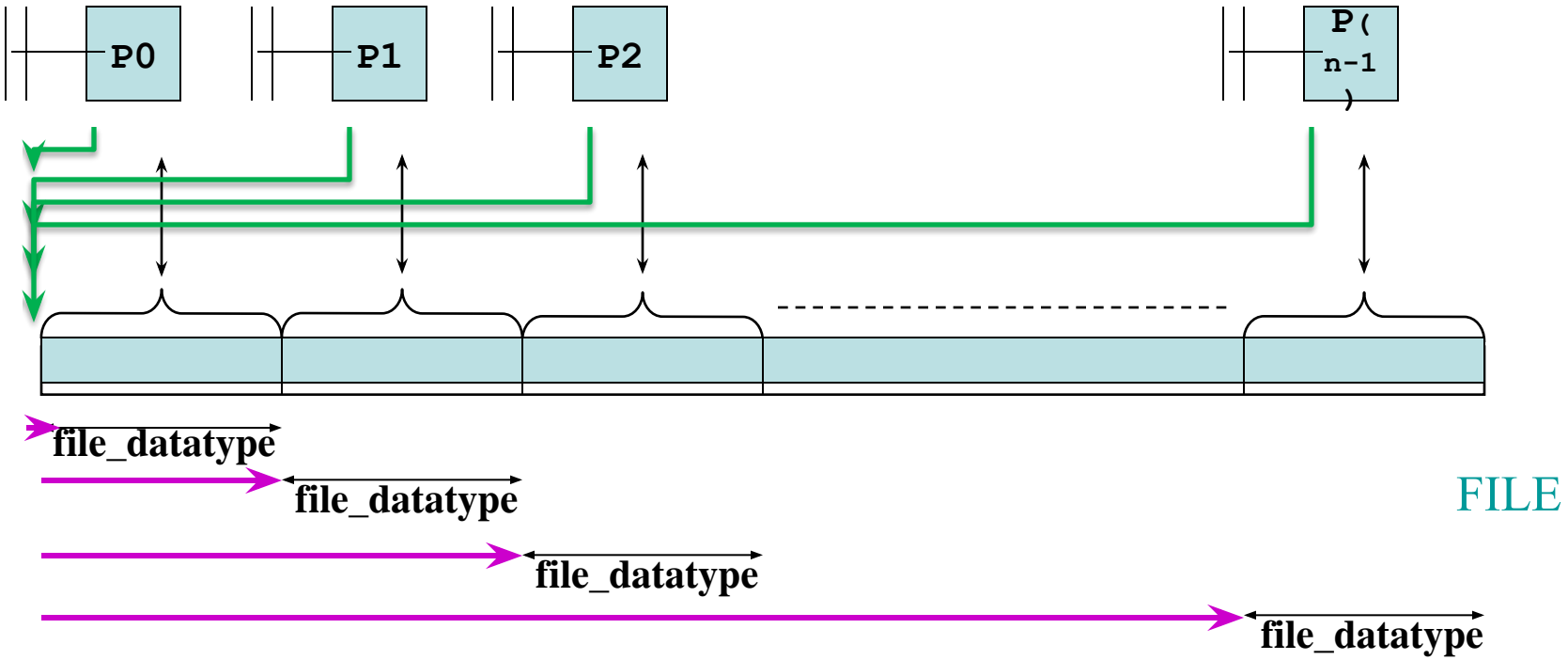
MPI_File_set_view(fd,…,disp, … file_datatype…)

MPI_File_write(fh, datatype ...)

Each process write (reads) a sequential block of data to (from) a single file

Blocked

Simple Approach: or Fileview Displacement (default pointers at 0)



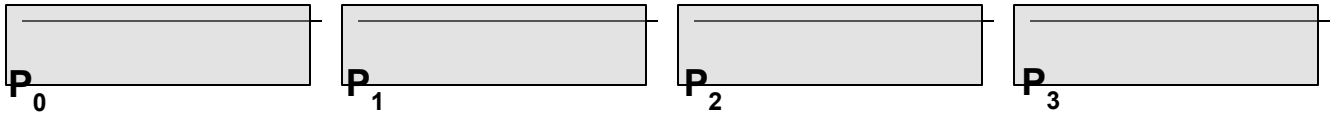
MPI_File_set_view(fd,…,disp, … file_datatype…)

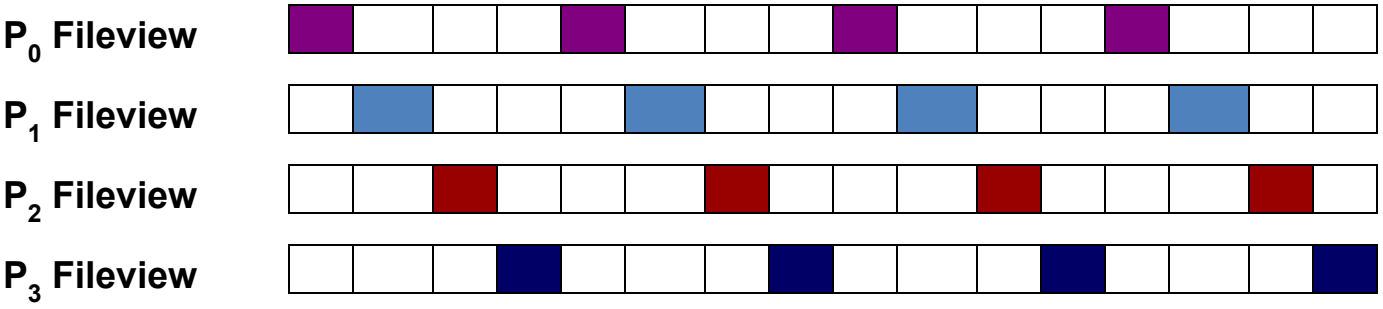
MPI_File_write(fh, datatype ...)

Each process write (reads) a sequential block of data to (from) a single file

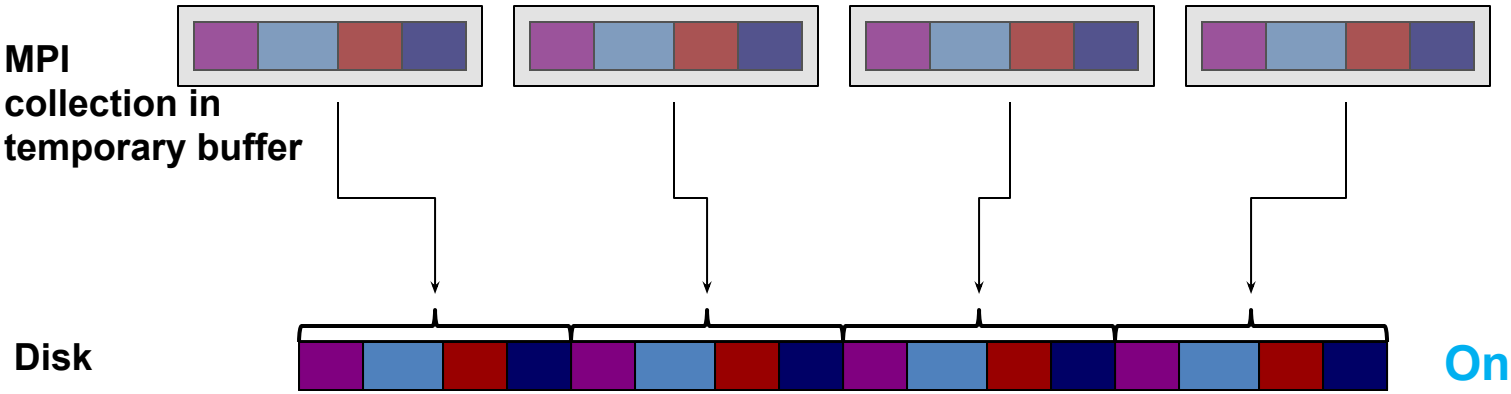
COLLECTIVE I/O

Memory layout on 4 processor





IO for each block (element) may be expensive.



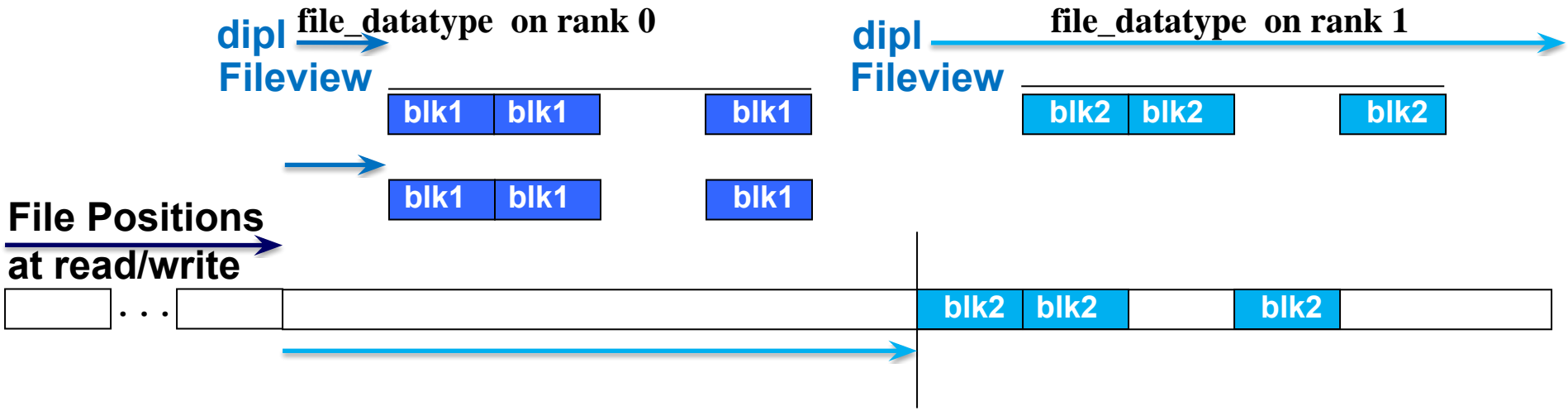
Collective IO Aggregates Write/Reads

Collective IO (mechanism will vary)

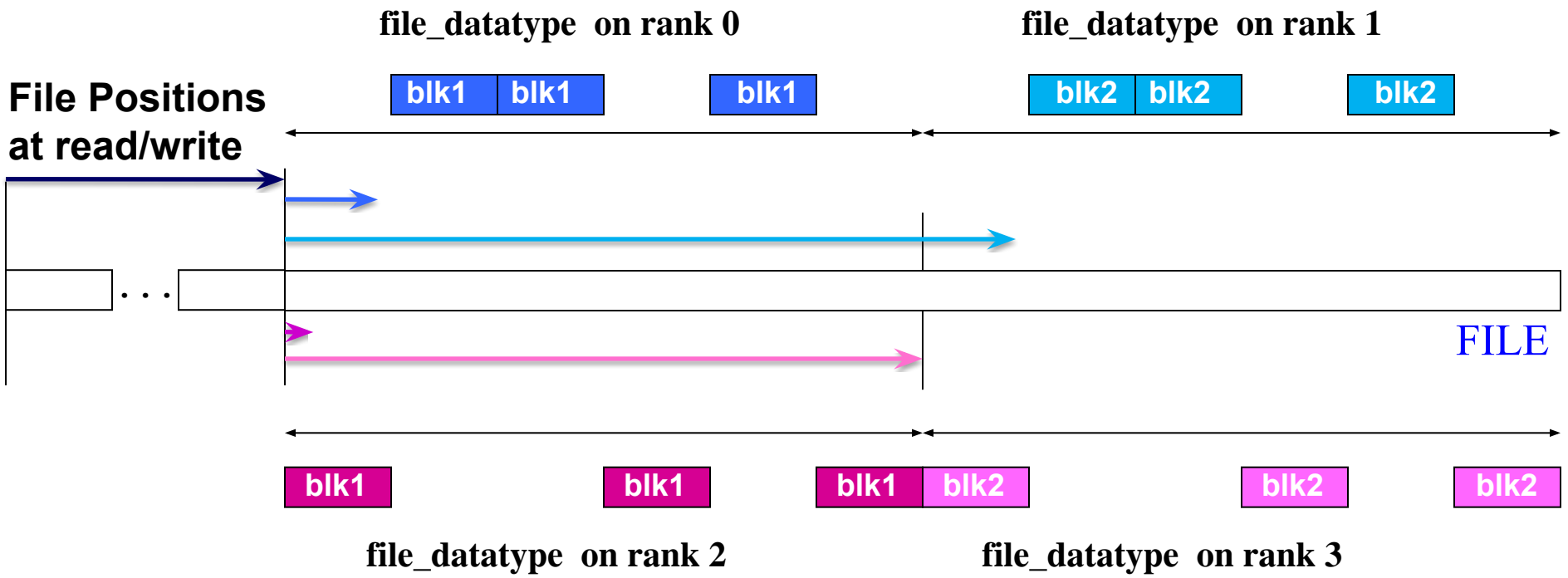
Only 4 operations Instead of 16.

More Complicated Layout

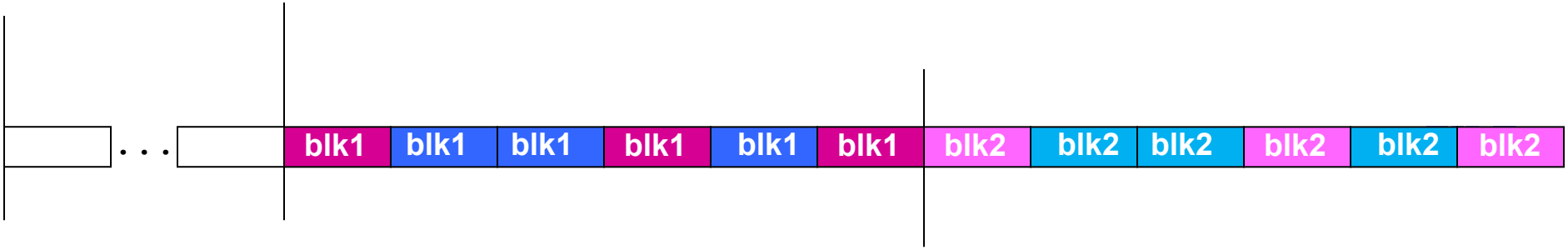
Open "myfile" file and get its filehandle(fd):
```
MPI_File_open(… "myfile", fd, …)
```
Create a vector, indexed, etc. data type:
```
MPI_Type_*(…, file_datatype, …)
```
Associate datatype with the filehandle (fd):
```
MPI_File_view (fd,
...file_datatype...)
```



Non-contiguous, multi-blocked



Non-contiguous, multi-blocked



Homework

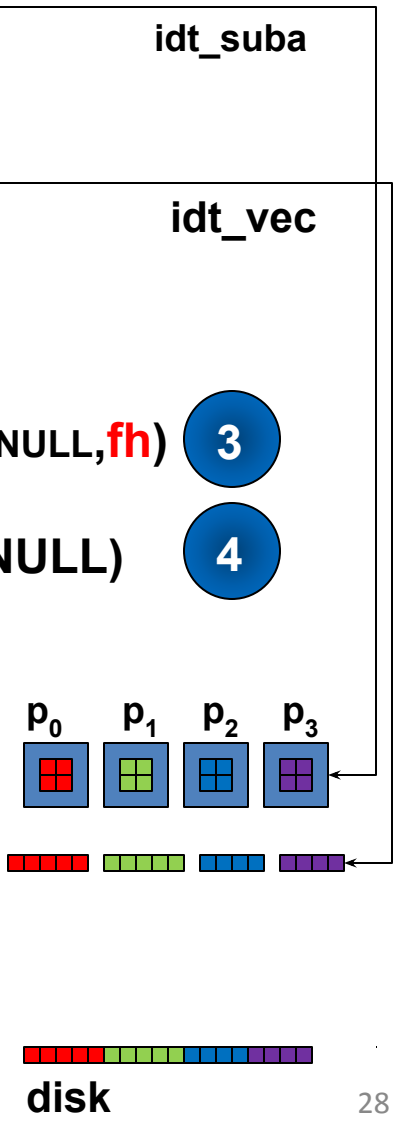
```
MPI_Type_create_subarray(…, idt_suba)   1
MPI_Type_vector  (…, idt_vec)           2

!                    consolidate File view info
etype   = MPI_REAL8
filetype= idt_vec
MPI_FILE_OPEN(MPI_COMM_WORLD,"data2", iamode,MPI_INFO_NULL,fh)   3

MPI_FILE_SET_VIEW(fh,disp,etype,filetype,"native",MPI_INFO_NULL)   4

MPI_FILE_WRITE(fh,data,     1,idt_suba   ,status,ierr)   5
```

1. Create a datatype for memory layout (a subarray)
2. Create a datatype for the file layout (a vector of only 1 block)
3. Open the file, note file handle (fh)
4. Tell MPI the datatype to be used on the disk when writing to “fh”.
5. Tell MPI to write “data” to file:
   extracting subarray from memory
   writing a idt_vec layout to disk.



Positioning	Synchronism	Coordination (noncollective)	Coordination (collective)
Explicit offsets	Blocking	read_at	read_at_all
		write_at	write_at_all
	Nonblocking & split collective	iread_at	read_at_all_begin
			read_at_all_end
		iwrite_at	write_at_all_begin
			write_at_all_end
Individual pointers	Blocking	read	read_all
		write	write_all
	Nonblocking & split collective	iread	read_all_begin
			read_all_end
		iwrite	write_all_begin
			write_all_end
Shared file pointer	Blocking	read_shared	read_ordered
		write_shared	write_ordered
	Nonblocking & split collective	iread_shared	read_ordered_begin
			read_ordered_end
		iwrite_shared	write_ordered_begin
			write_ordered_end

MPI-IO Open

Use MPI MPI_FILE_OPEN call to open file.

C

```
MPI_File_open(comm, filename, mode, info, &fh)
```

FORTRAN

```
MPI_File_open(comm, filename, mode, info,  fh,
ierr)
```

argument	function	C	Fortran
comm	Communicator	MPI_Comm	integer
filename	filename	char*	character*(*)
mode	access mode	int	integer
info	info handle	MPI_Info	integer
fh	file handle	MPI_File	integer
ierr	error number	--	integer

MPI-IO Open Access

The file access (combine with "or")

```
MPI_MODE_RONLY            - read  only
MPI_MODE_WRONLY           - write only
MPI_MODE_RDWR             - read and write
MPI_MODE_CREATE           - create the file if it does not exist
MPI_MODE_EXCL             - error on creating file that exists
MPI_MODE_DELETE_ON_CLOSE - delete file on close
MPI_MODE_UNIQUE_OPEN      - file is not concurrently opened
MPI_MODE_SEQUENTIAL       - file will only be accessed sequentially
MPI_MODE_APPEND - set initial position of all file pointers to EOF
```

```
e.g.
F90: mode=ior(MPI_MODE_CREATE,  MPI_MODE_WRONLY)
C:   mode=   (MPI_MODE_CREATE | MPI_MODE_WRONLY)
```

Examples

Parallel I/O with Individual File Pointers

```
MPI_File fh;                    int chunk,count,buf[ ];
MPI_Status status;
MPI_Offset offset
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

chunk   = FILESIZE/npes;
count   = chunk/sizeof(int); //care with RHS of offset
offset  = rank * bufsize      //default units: MPI_BYTE

MPI_File_open(MPI_COMM_WORLD, "/work/me/myfile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh,offset,MPI_SEEK_SET);
MPI_File_read(fh,buf,count, MPI_INT,&status);
MPI_File_close(&fh);
```

Using Explicit Offsets

```
include 'mpif.h'
integer :: fh,ierr,count,offset,rank,INTSIZE=4,buf()
integer                              :: status(MPI_STATUS_SIZE)
integer(kind=MPI_OFFSET_KIND) :: offset

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

call MPI_File_open(MPI_COMM_WORLD, "/dir/datafile", &
                   MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)

call MPI_File_read_at(fh, offset, buf, count,  &
                      MPI_INTEGER, status, ierr)

call MPI_Get_count(status, MPI_INTEGER, count, ierr)

print *, "process,# of ints", rank, count
call MPI_FILE_CLOSE(fh, ierr)
```

MPI-IO with a View

Form an efficient strategy of how each process will perform IO to a disk– this will be the basis for each task's "view" of the disk data.

- Open file
- Create a datatype (filetype) for viewing disk for each process.
- In MPI_File_set_view set: displacement, etype, and filetype*
- Position file pointer (or not) and transfer data in parallel
- Flush data buffers
- Close file

***Default view: displacement = 0, etype=filetype=MPI_BYTE**

MPI-IO View

Set view with MPI_File_set_view.

C

```
MPI_File_set_view(fh, disp, etype, filetype, &datarep, info)
```

FORTRAN

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info,ierr)
```

Argument	Function	C	Fortran
fh	file handle	MPI_File	integer
disp	displacement	MPI_Offset	integer(MPI_OFFSET_KIND)
etype	unit of access	MPI_Datatype	integer
filetype	repeated unit	MPI_Datatype	integer
datarep	data-representation	char *	character*(*)
info	info handle	MPI_Info	integer
ierr	error number	--	integer

Calling MPI_File_set_view sets the shared and individual file pointers to 0.

disp is in bytes, but offset in MPI_File_xxx_at is counts of etype (MPI_Offset / MPI_OFFSET_KIND)

A filetype is either an etype or a derived MPI datatype of multiple etypes.

File View Example

```
#include <mpi.h>
...

    MPI_File     fh;
    MPI_Status   status;
    MPI_Offset   disp;
    MPI_Datatype etype, filetype;
    int rank,mode;

    int intsize,i,num;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    mode=(MPI_MODE_CREATE |
MPI_MODE_WRONLY);
```

File View Example

```
MPI_File_open(MPI_COMM_WORLD,"data", mode,MPI_INFO_NULL, &fh);

/* MPI_Type_extent(MPI_INT,&intsize);
   disp=(MPI_Offset)3*rank*intsize */

disp=(MPI_Offset)3*rank*sizeof(int);
etype   =MPI_INT;
filetype=MPI_INT;
```

"external" = XDR (portable)
"external32" IEEE big-endian"

```
MPI_File_set_view(fh,disp,etype,filetype,"native",MPI_INFO_NULL);

 for(i=1;i<4;i++){
   num=i + (rank+1)*100;
   MPI_File_write(fh,&num,1,MPI_INT,&status);
 }
 MPI_File_sync(  fh);
 MPI_File_close(&fh);
}
```

Regularly distributed arrays

```
#include "mpi.h"
#include <stdio.h>
#define N 3
int main(int argc, char *argv[])
{
  int ierr, iam, np;
  MPI_Comm MCW;
  MPI_Init(&argc,&argv);
  MPI_Comm_dup(MPI_COMM_WORLD,&MCW);
  ierr=MPI_Comm_size(MCW,&np);
  ierr=MPI_Comm_rank(MCW,&iam);
  double data[N];
  MPI_File fh;
```

Regularly distributed arrays

```
  ierr = MPI_File_open(MCW,"test.out",
                       MPI_MODE_WRONLY|MPI_MODE_CREATE,
                       MPI_INFO_NULL,&fh);
  for(int i=0; i<N; ++i)
    data[i]=(double)(iam);
  MPI_Barrier(MCW);
  MPI_Datatype filetype;
  ierr=MPI_Type_vector(N,1,np,MPI_DOUBLE,&filetype);
  ierr=MPI_Type_commit(&filetype);
  ierr=MPI_File_set_view(fh,sizeof(double)*iam,MPI_DOUBLE,
                         filetype,"native",MPI_INFO_NULL);
  MPI_Status status;
  ierr=MPI_File_write_all(fh,data,N,MPI_DOUBLE,&status);
  MPI_File_close(&fh);
  MPI_Finalize();
  return(0);
}
```