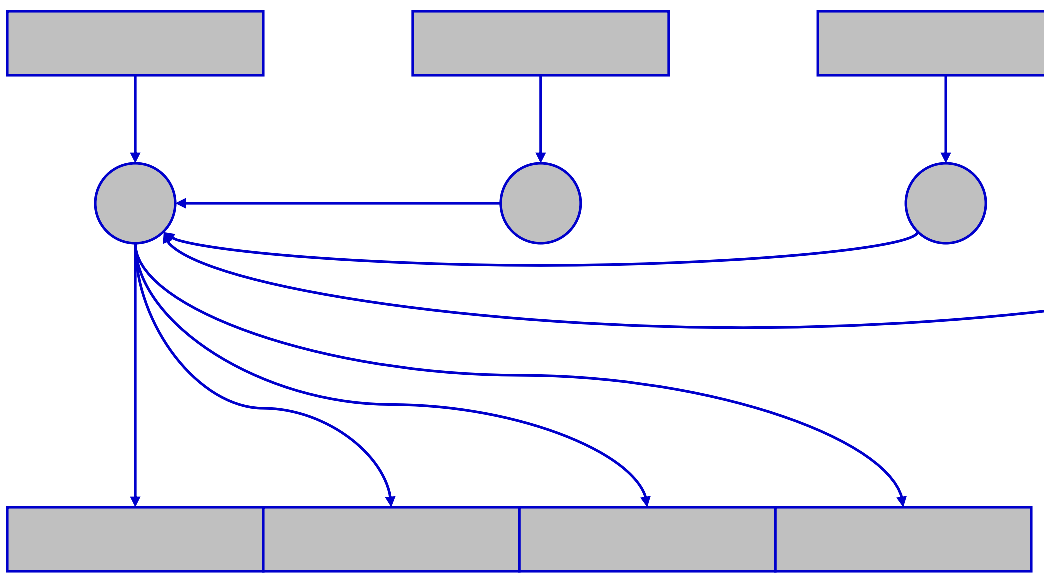
Common Ways of Doing I/O in Parallel Programs

- Sequential I/O:
  - All processes send data to rank 0, and 0 writes it to the file

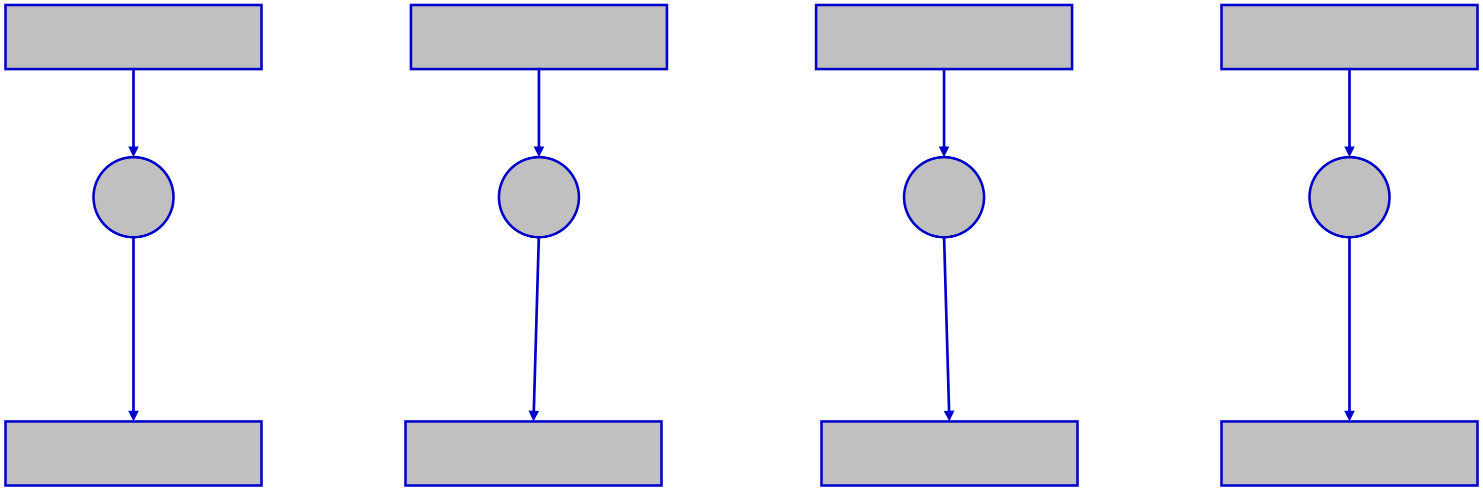


Pros and Cons of Sequential I/O

- Pros:
  - parallel machine may support I/O from only one process (e.g., no common file system)
  - Some I/O libraries not parallel
  - resulting single file is handy for **`ftp`**, **`mv`**
  - big blocks improve performance
  - short distance from original, serial code
- Cons:
  - lack of parallelism limits scalability, performance (single node bottleneck)

Another Way

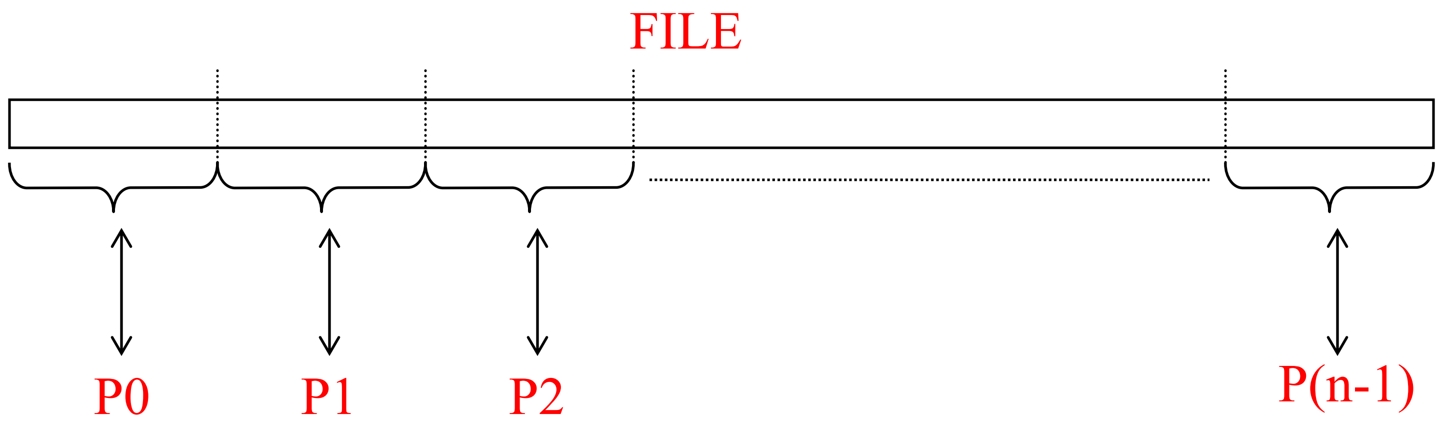
- Each process writes to a separate file



- Pros:
  - parallelism, high performance
- Cons:
  - lots of small files to manage
  - difficult to read back data from different number of processes

What is Parallel I/O?

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



Why Parallel I/O?

- Non-parallel I/O is simple but
  - Poor performance (single process writes to one file) or
  - Awkward and not interoperable with other tools (each process writes a separate file)
- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

Why is MPI a Good Setting for Parallel I/O?

- Writing is like sending a message and reading is like receiving.
- Any parallel I/O system 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*)

MPI API

Constructor	Purpose
MPI_File_open()	Opens a file on all processes in the communicator group
MPI_File_close()	Closes a file on all processes in the communicator group
MPI_File_delete()	Deletes a file
MPI_File_write() MPI_File_write_all() MPI_File_write_ordered() MPI_File_write_at() MPI_File_write_shared()	Write using individual file pointer; Collective write using individual file pointer; Collective write using shared file pointer; Write using explicit offset. Write using shared file pointer
MPI_File_read() MPI_File_read_all() ...	Read using individual file pointer; Collective read using individual file pointer; ...
MPI_File_seek()	Updates the individual file pointer
MPI_File_set_view()	Changes the process's view of the data in the file

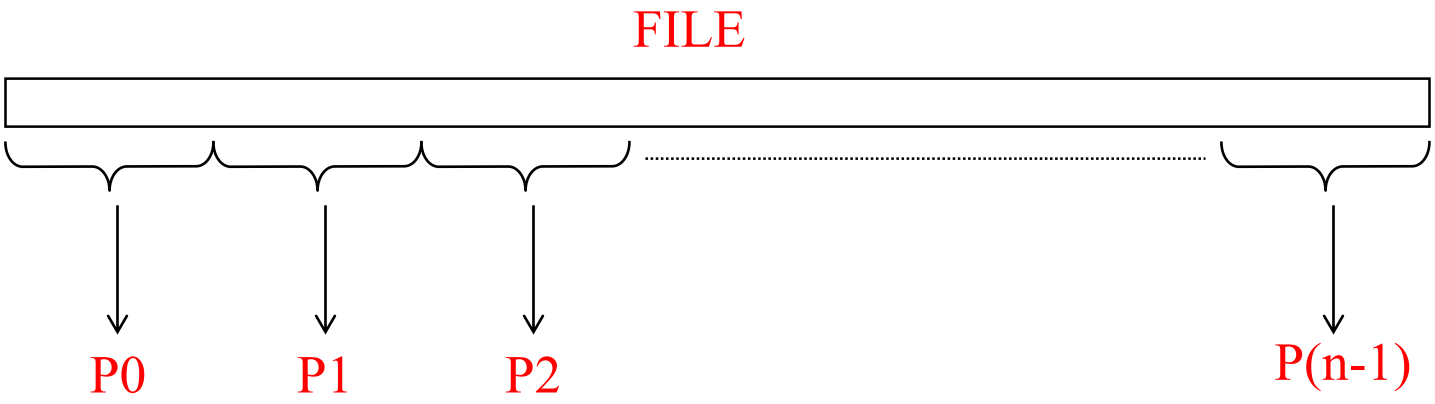
Shared and Individual File Pointers

- MPI allows reading / writing of files using two different kinds of file pointers:
  - Shared file pointer: file pointer is shared among all processes in the communicator used to open the file. Same pointer for all processors.
    - Only one processor can “own” shared pointer for writing or reading at a time.
  - Functions are collective
  - Examples: MPI_Write_shared(), MPI_Write_ordered(), MPI_File_seek_shared() and the corresponding MPI_Read_...() functions.

Shared and Individual File Pointers

- Individual file pointer: each process has its own local file pointer for seek, read and write operations
  – Non-collective version (e.g. MPI_File_write(), MPI_File_read());
  – Collective version (e.g. MPI_File_write_all()): generally more efficient
- Finally, there’s the concept of file view: maps data from multiple processors to the file representation on disk.

Using MPI for Simple I/O



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

I/O Using Shared Pointers

- The function MPI_Write_ordered() provides a collective access using a shared file pointer
- Accesses to the file will be in the order determined by the ranks of the processes within the group
- Reading done using the corresponding function MPI_File_read_ordered().
  - For each process, the access location in the file is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data.

I/O Using Individual Pointers

- The same result can be obtained using a combination of MPI_File_seek() and MPI_File_write()
- The function MPI_File_write() does the writing at the file pointer position
  - MPI_File_write() is non-collective
  - MPI_File_write_all() is collective version

Using Individual File Pointers

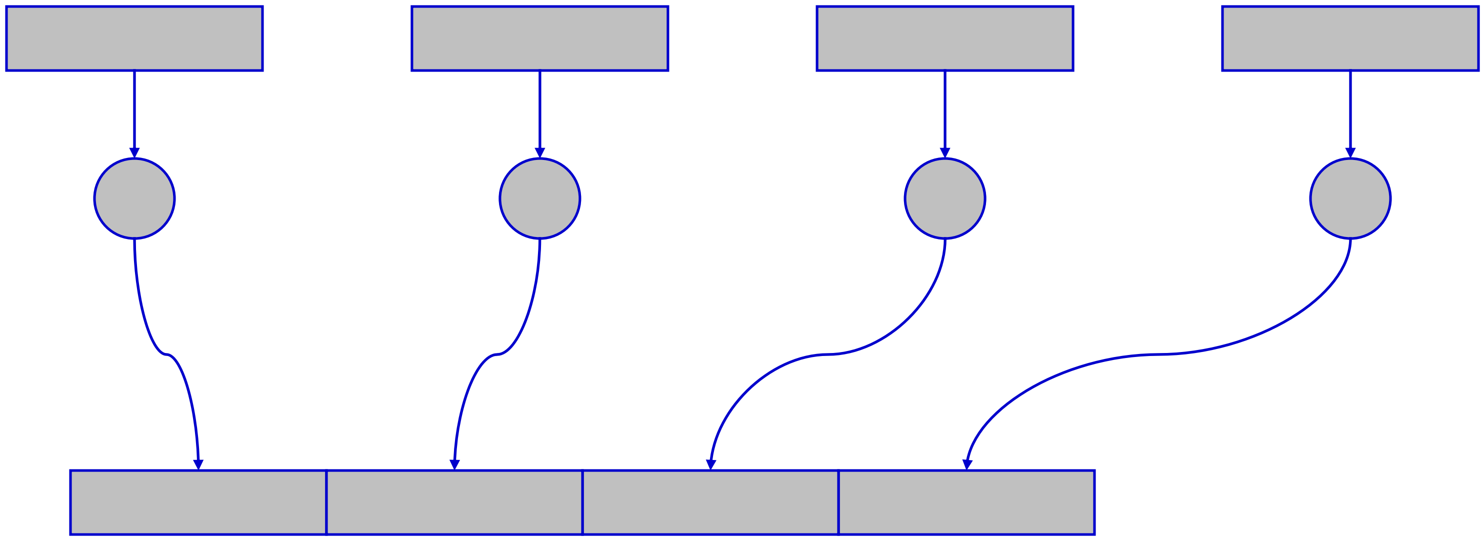
```
MPI_File fh;
MPI_Status status;

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);
```

Writing to a File

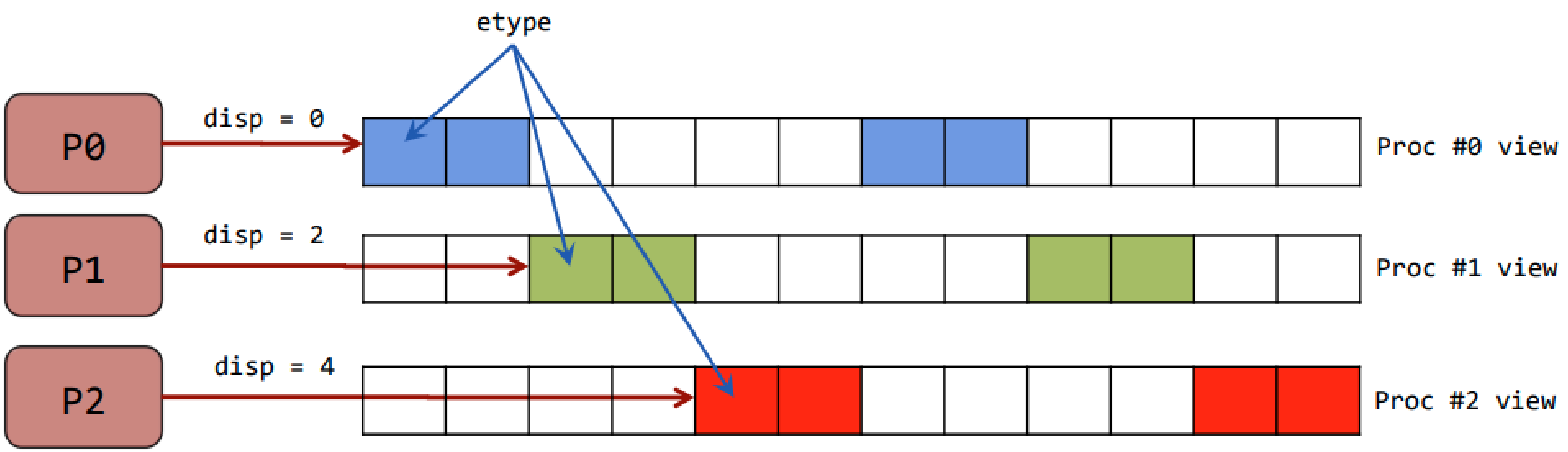
- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- Use **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** as the flags to **`MPI_File_open`**
- If the file doesn't exist previously, the flag **`MPI_MODE_CREATE`** must also be passed to **`MPI_File_open`**
- We can pass multiple flags by using bitwise-or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


etype
P0
disp = 0
Proc #0 view
P1
disp = 2
Proc #1 view
P2
disp = 4
Proc #2 view

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;



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`**
  - **`MP_Offset disp`**
  - **`MPI_Datatype etype`**
  - **`MPI_Datatype filetype`**
  - **`char *datarep`**
  - **`MPI_Info info`**