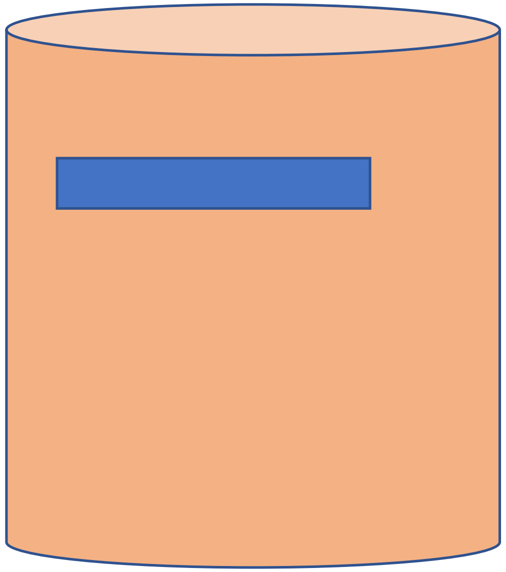
Parallel I/O - I

Lecture 15

March 11, 2024

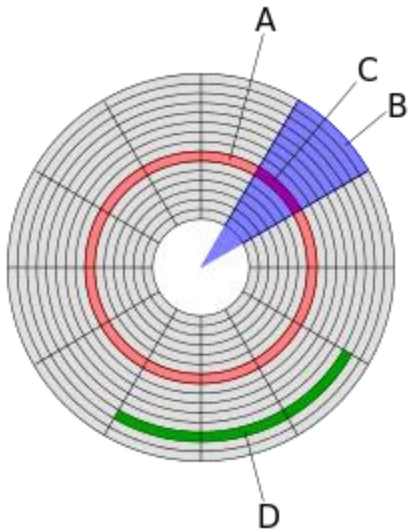
Sequential File Handling

```
char mystring[] = “Hello world”
FILE *fp = fopen (“/data/smallfile”, “w”)
fwrite (mystring,  sizeof(char), sizeof(mystring), fp)
fclose (fp)
```



Hard Disk Drive

- One process reads/writes to a file
- Files are stored on hard disk drives
  - Rotating disks
  - Read/write heads
  - Sequential access
  - Seek time + Rotational latency



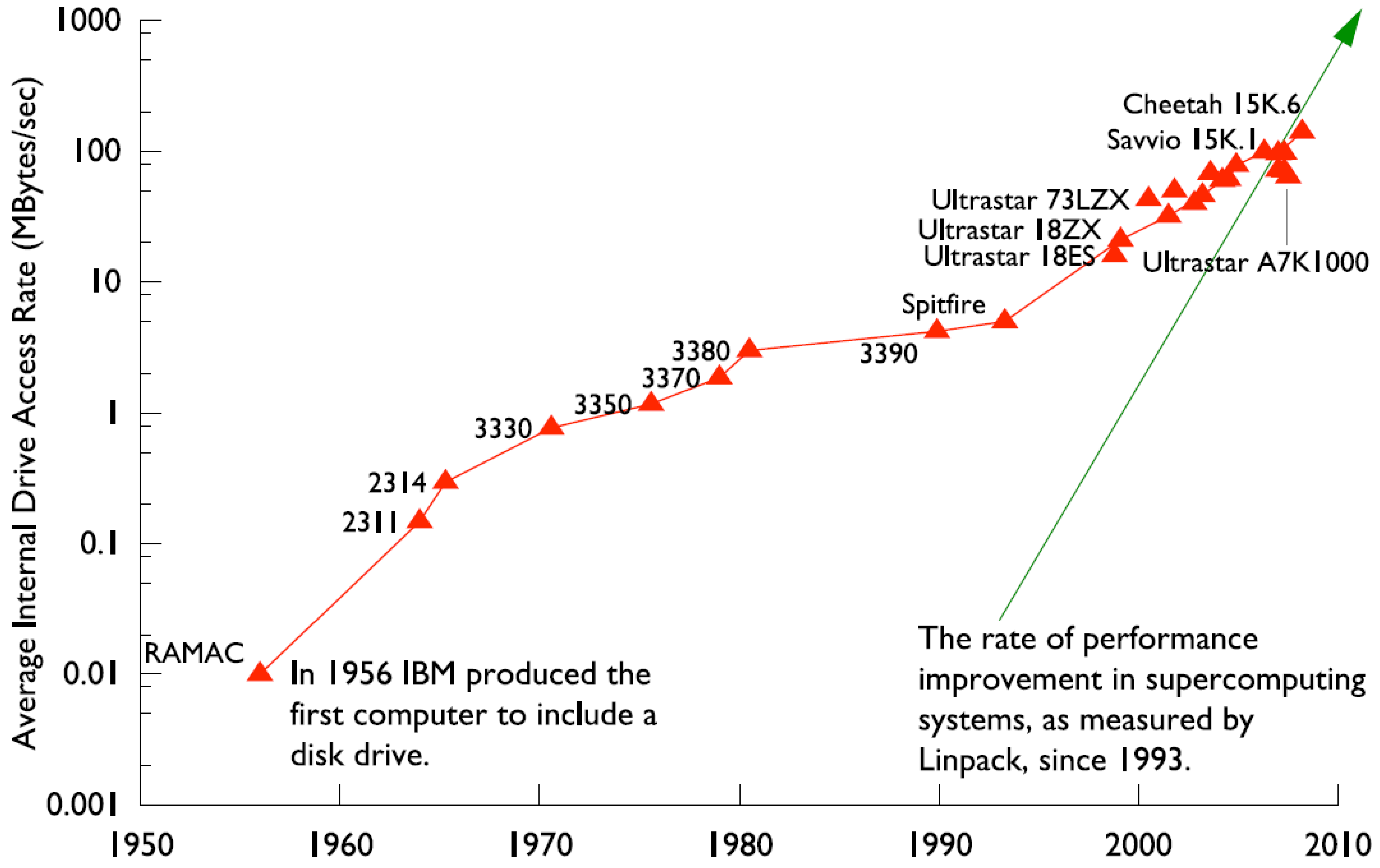
A. Track
B. Geometrical sector
C. Track sector
D. Cluster

[Source: Wikipedia]

- Mechanical device
- Magnetic storage medium
- Primary persistent storage device

https://www.youtube.com/watch?v=sG2sGd5XxM4

Disk Access Rates



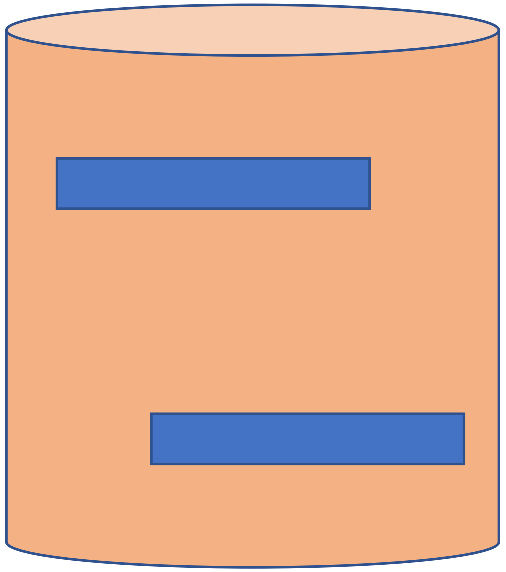
[Credit: R. Ross, ANL and R. Freitas, IBM]

Storage Devices

Access speed (I/O w bandwidth)

- HDD
- SSD
- NVRAM
- RAM

- ~ 300 MB/s
- ~ 800 MB/s
- ~ 1 - 2 GB/s
- ~ 1 GB/s



I/O Bandwidths

```
pmalakar@csews5:~$ dd of=/dev/zero if=testfile bs=1K count=1000K
1024000+0 records in
1024000+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 1.42756 s, 735 MB/s
pmalakar@csews5:~$
pmalakar@csews5:~$
pmalakar@csews5:~$ dd of=/dev/zero if=testfile bs=1K count=1000K
1024000+0 records in
1024000+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 1.29833 s, 808 MB/s
pmalakar@csews5:~$
```

Read bandwidth

```
1024000+0 records in
1024000+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 12.4325 s, 84.3 MB/s
pmalakar@csews5:~$
pmalakar@csews5:~$ dd if=/dev/zero of=/tmp/testfile bs=1K count=1000K
1024000+0 records in
1024000+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 2.42031 s, 433 MB/s
pmalakar@csews5:~$ dd if=/dev/zero of=/tmp/testfile bs=1K count=1000K
1024000+0 records in
1024000+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 7.34824 s, 143 MB/s
```

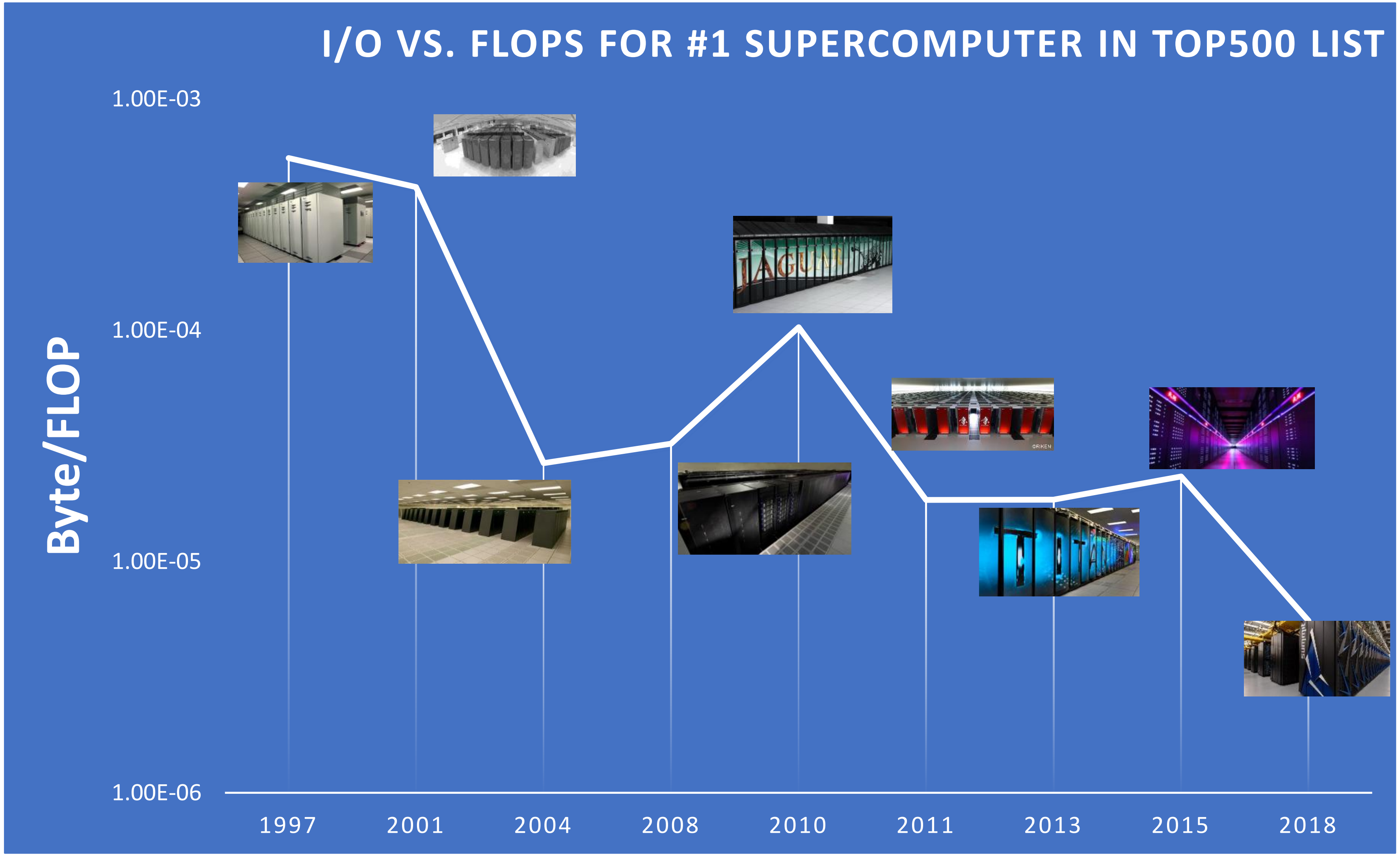
Write bandwidth

Data Requirements

Application	Data Requirements
FLASH: Buoyancy-Driven Turbulent Nuclear Burning	300 TB
Reactor Core Hydrodynamics	5 TB
Computational Protein Structure	1 TB
Kinetics and Thermodynamics of Metal and Complex Hydride Nanoparticles	100 TB
Climate Science	345 TB
Parkinson's Disease	50 TB
Lattice QCD	44 TB

[Source: 2008 report, S. Klasky]

Compute vs. I/O trends



Parallel I/O

What?

- Every process reads and writes files in parallel
- Simultaneous access to storage (at least the illusion of it)

Why?

- Input/output data is of the order of TBs!
- Disk access rates are of the order of GB/s
- Speed up data availability in the process’ memory

Write to Same File

File pointer

Solution?

Uncoordinated writes

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>
#define BUFSIZE 10000

int main(int argc, char *argv[]) {

        int i, myrank, buf[BUFSIZE];
        char filename[128];
        FILE *myfile;

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

        strcpy(filename, "testfile");
        myfile = fopen(filename, "w");

        for (i=0; i<BUFSIZE; i++) {
           buf[i] = myrank + i;
           fprintf(myfile, "%d\n", buf[i]);
        }
        fclose(myfile);

        MPI_Finalize();
        return 0;

}
```

Write to Different Files

Independent writes

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 1000

int main(int argc, char *argv[]) {

        int i, myrank, buf[BUFSIZE];
        char filename[128];
        FILE *myfile;

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

        sprintf(filename, "testfile.%d", myrank);
        myfile = fopen(filename, "w");

        for (i=0; i<BUFSIZE; i++) {
           buf[i] = myrank + i;
           fprintf(myfile, "%d ", buf[i]);
        }
        fprintf(myfile, "\n");
        fclose(myfile);

        MPI_Finalize();
        return 0;

}
~
```

Timing (Different data sizes for P=1 and P=4)

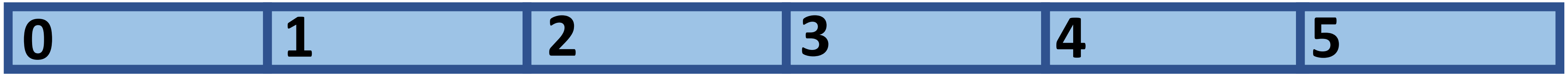
0.0225s	0.0841s
1.911s	4.251s
23.779s	46.416s

Q: Problem with writing to different files per rank?

Simple Parallel I/O Code – Shared File

```
MPI_File fh
file_size_per_proc = FILESIZE / nprocs
MPI_File_open (MPI_COMM_WORLD, “/scratch/largefile”,
MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)
MPI_File_seek (fh, rank*file_size_per_proc, MPI_SEEK_SET)
MPI_File_read (fh, buffer, count, MPI_INT, status)
MPI_File_close (&fh)
```

Returns file handle



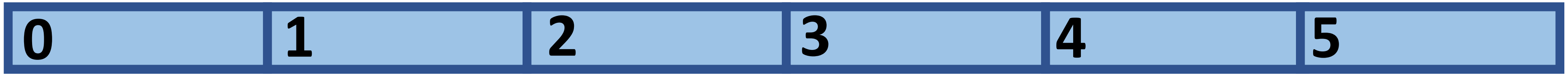
Parallel Read using Explicit Offset

MPI_Offset offset = (MPI_Offset) rank*file_size_per_proc*sizeof(int)

MPI_File_open (MPI_COMM_WORLD, “/scratch/largefile”, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)

MPI_File_read_at (fh, offset, buffer, count, MPI_INT, status)

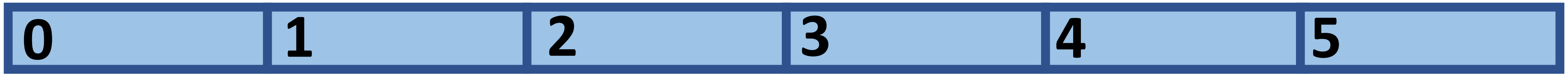
MPI_File_close (&fh)



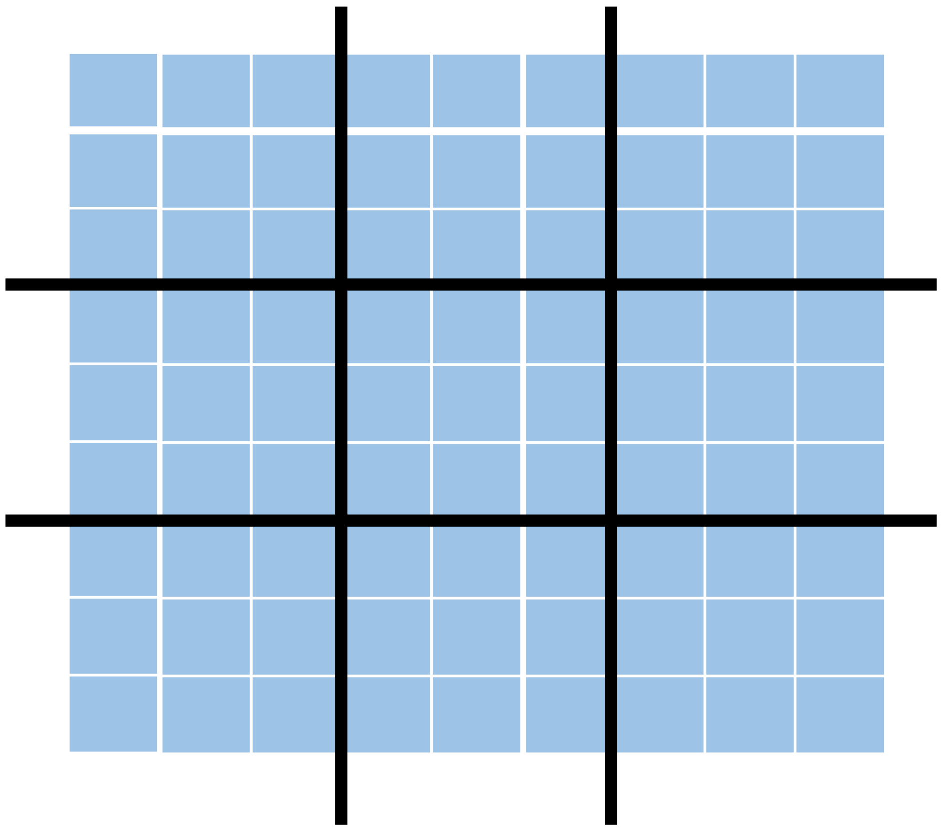
```c
MPI_File fh;  // FILE

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

for (int i=0; i<BUFSIZE ; i++)
  buf[i]=i;

strcpy(filename, "testfileIO");

// File open, fh: individual file pointer
MPI_File_open (MPI_COMM_WORLD, filename, MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

MPI_Offset fo = (MPI_Offset) myrank*BUFSIZE*sizeof(int);

// File write using explicit offset (independent I/O)
MPI_File_write_at (fh, fo, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE); //fwrite

// File read using explicit offset (independent I/O)
MPI_File_read_at (fh, fo, rbuf, BUFSIZE, MPI_INT, &status);   //fread

MPI_File_close (&fh);         //fclose

for (i=0; i<BUFSIZE ; i++)
  if (buf[i] != rbuf[i]) printf ("Mismatch [%d] %d %d\n", i, buf[i], rbuf[i]);
```

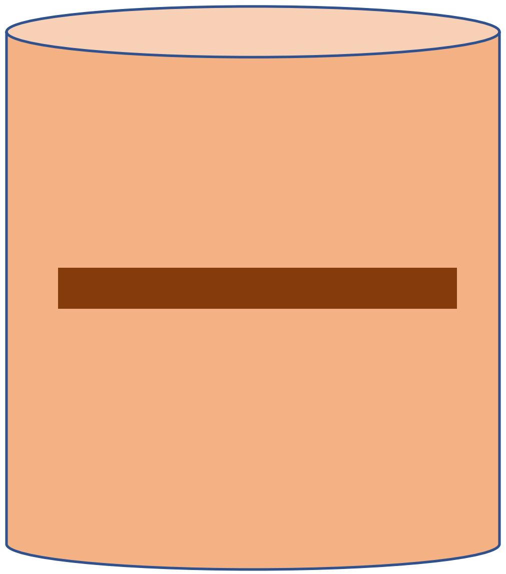
Features Summary

- Multiple processes access a common file
- Multiple processes access the same file at the same time
- Multiple seeks issued at the same time
- Each process reads a contiguous chunk
- Individual file pointers

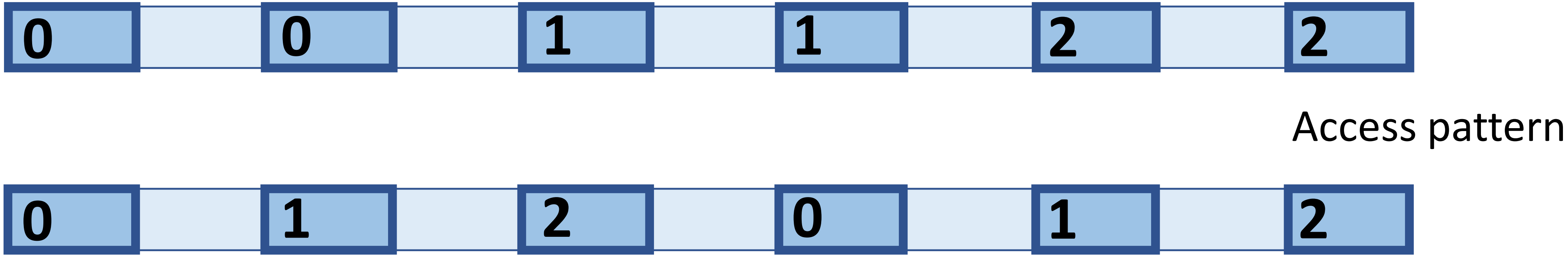


Large Domain





Non-contiguous Accesses

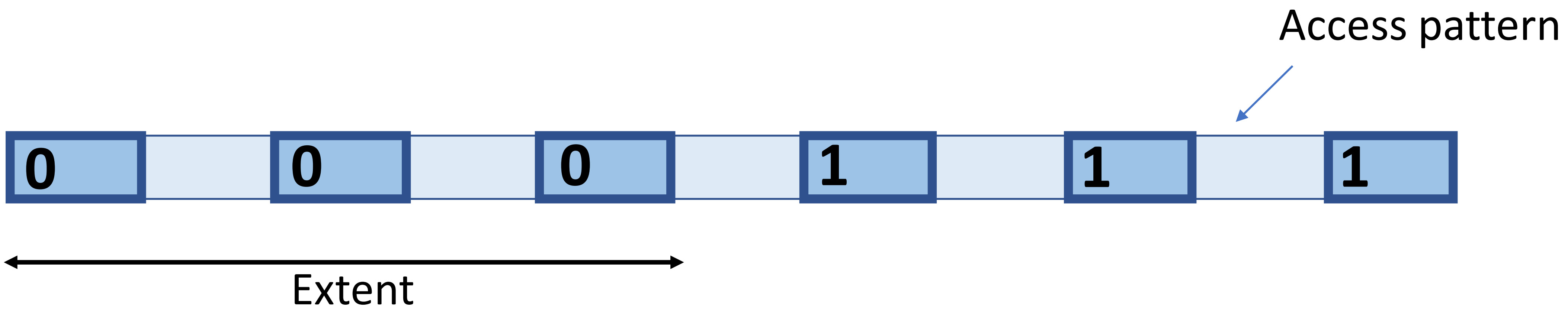


Frequently occurring file access pattern (non-contiguous)

What would be the required function calls?

What is the problem with non-contiguous accesses?

Multiple Short Accesses



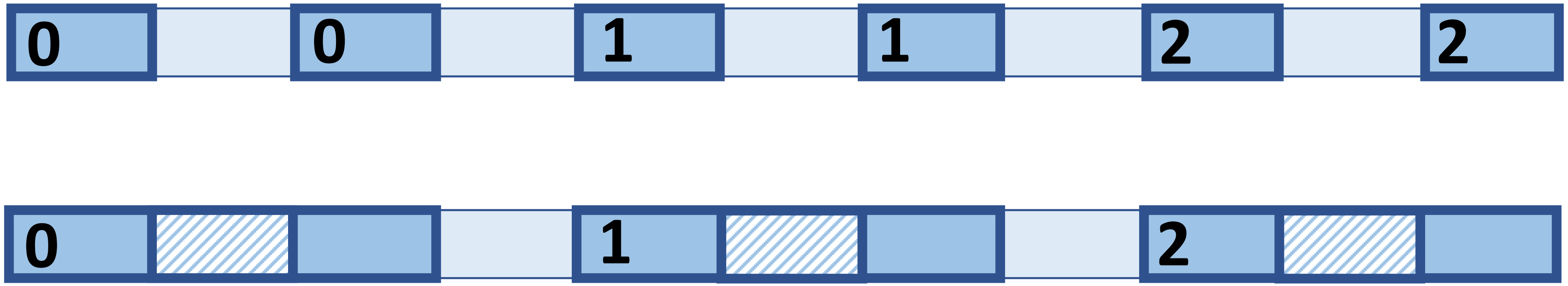
MPI_File_read_at (fh, offset1, buffer1, count1, MPI_INT, status)

MPI_File_read_at (fh, offset2, buffer2, count2, MPI_INT, status)

MPI_File_read_at (fh, offset3, buffer3, count3, MPI_INT, status)

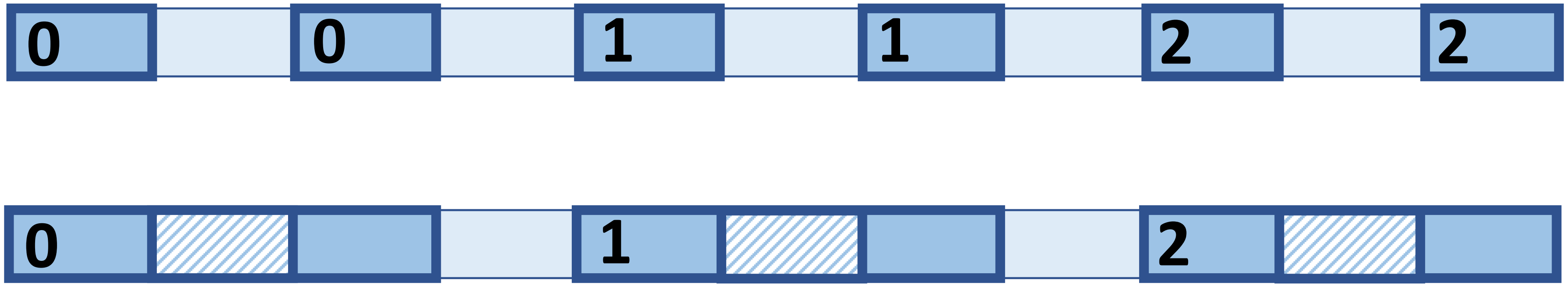
Can we instead use one read call?

Optimization – Data Sieving



- Make large I/O requests and extract the data that is really needed
- Huge benefit of reading large, contiguous chunks

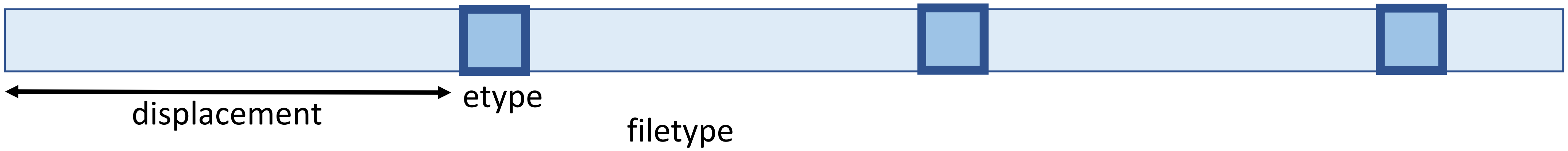
Data Sieving for Writes



- Copy only the user-modified data into the write buffer
- Write only the data that was modified – read-modify-write

File View

Non-contiguous access pattern can be specified using a view



Each process can specify a view

displacement
etype
filetype

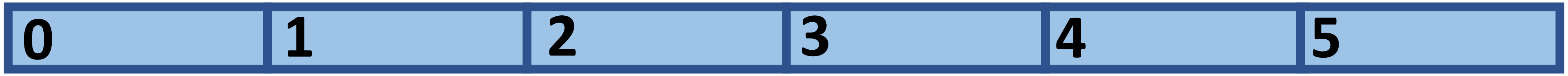
File view

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

MPI_File_read (fh, buffer, count, MPI_INT, status)

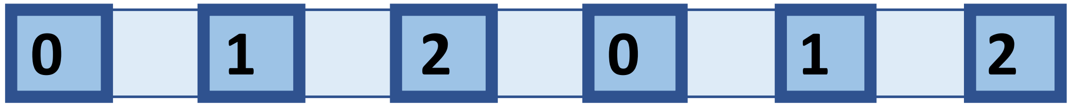
Independent I/O - Set File View

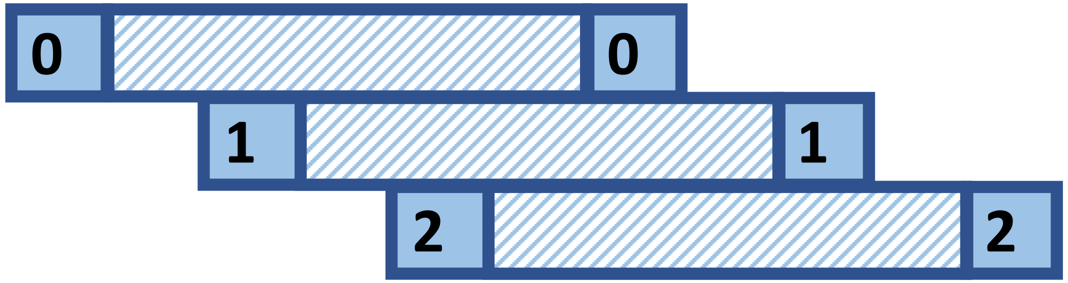
```
MPI_File_open (MPI_COMM_WORLD, filename, MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);

for (i=0; i<BUFSIZE; i++) {
   buf[i] = myrank + i;
}

// File write - set process view
MPI_File_set_view(myfile, myrank * BUFSIZE * sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write (myfile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);

// File read - set process view
MPI_File_set_view(myfile, myrank * BUFSIZE * sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_read (myfile, rbuf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);

MPI_File_close (&myfile);

for (i=0; i<BUFSIZE; i++) {
  if (buf[i] != rbuf[i]) printf ("%d %d %d\n", i, buf[i], rbuf[i]);
}
```



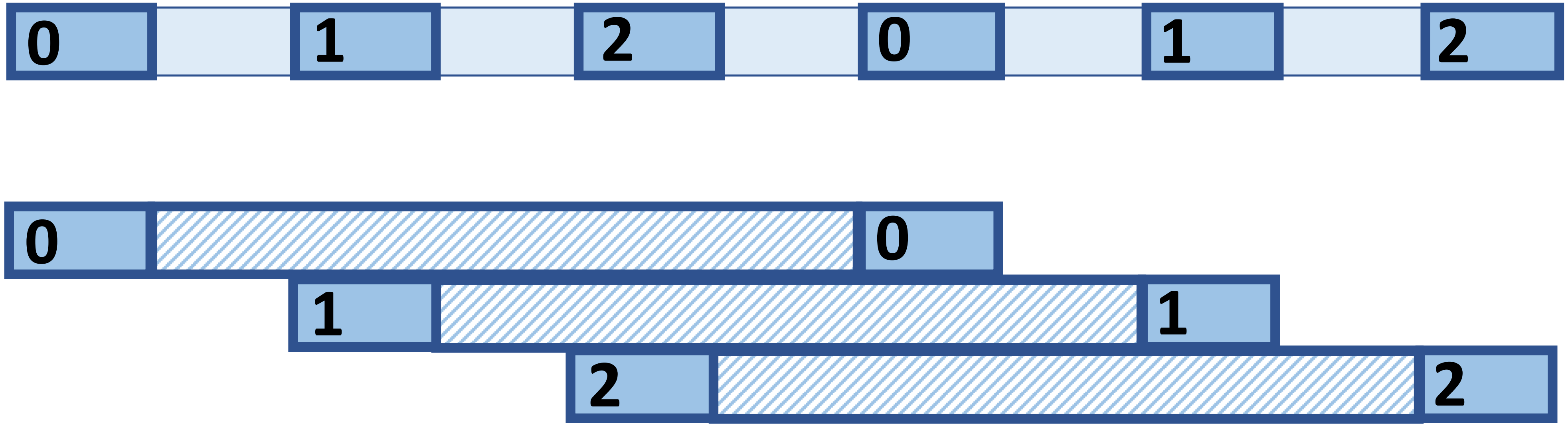
Non-contiguous Access – Set File View

MPI_Init
MPI_File_open
MPI_Type_vector
MPI_Type_commit
MPI_File_set_view
MPI_File_read
MPI_File_close
MPI_Type_free
MPI_Finalize





Data Sieving – Interleaved data



Q: What is the problem here (writes)?

Solution – Lock the relevant portions in the file

User-controlled MPI-IO Parameters

- ind_rd_buffer_size - Buffer size for data sieving for read
- ind_wr_buffer_size - Buffer size for data sieving for write
- romio_ds_read - Enable or not data sieving for read
- romio_ds_write - Enable or not data sieving for write

MPI_Info – Example

```
MPI_Info_create (&info);
MPI_Info_set (info, "ind_rd_buffer_size", "2097152");
MPI_Info_set (info, "ind_wr_buffer_size", "1048576");

MPI_File_open (MPI_COMM_WORLD, filename, amode, info, &fh);
```

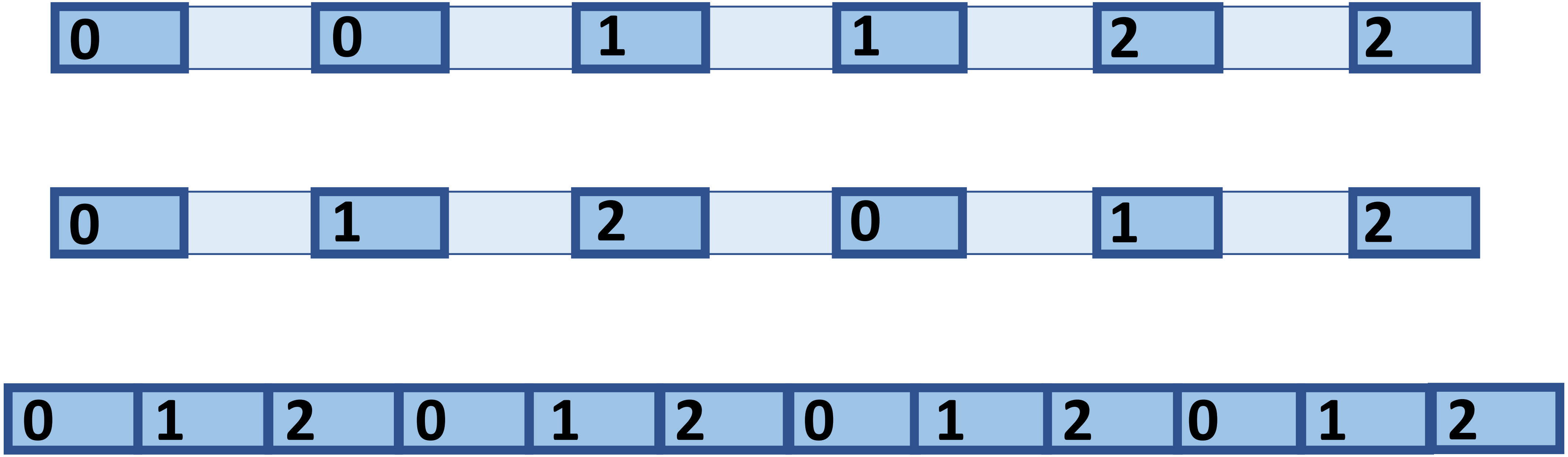
Parallel I/O Classification

- Independent I/O (we saw till now)

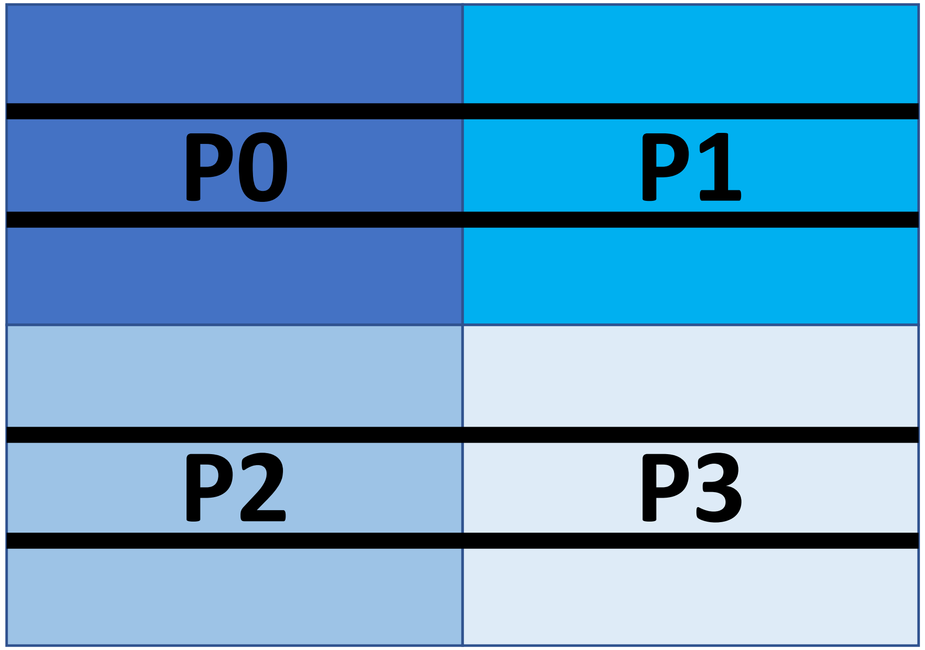
  MPI_File_read_at/MPI_File_read

- Collective I/O (we will see next)

Non-contiguous Accesses



Multiple Non-contiguous Accesses



- Every process' local array is non-contiguous in file
- Every process needs to make small I/O requests
- Can these requests be merged?

MPI Collective I/O

MPI_File_open (MPI_COMM_WORLD, “/scratch/largefile”, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)

MPI_File_read_at_all (fh, offset, buffer, count, MPI_INT, status)

or

MPI_File_set_view ….

MPI_File_read_all (fh, buffer, count, MPI_INT, status)

MPI_File_close (&fh)

```
for i in `seq 1 2` ; do mpirun -np 4 ./indep 16384 ; done
time diff 0.020404 3.063181 MB/s
time diff 0.021844 2.861241 MB/s
```

```
for i in `seq 1 2` ; do mpirun -np 4 ./coll 16384 ; done
time diff 0.004425 14.124899 MB/s
time diff 0.002833 22.062279 MB/s
```

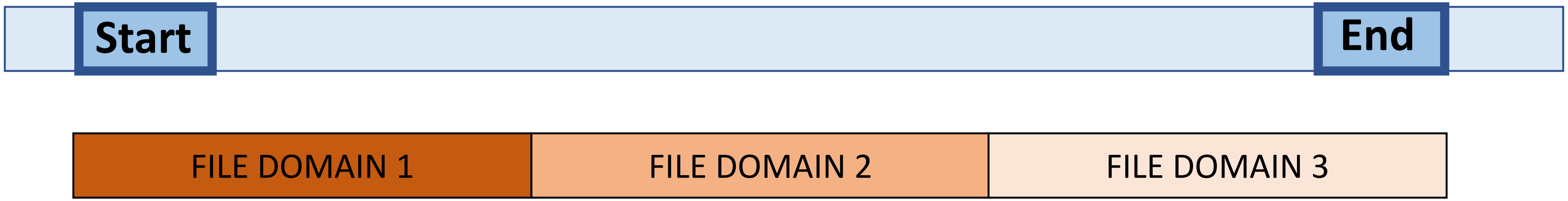
Two-phase I/O

Entire access pattern must be known before making file accesses

- Phase 1
  - Processes request for a single large contiguous chunk
  - Reduced file I/O cost due to large accesses
- Phase 2
  - Processes redistribute data among themselves
  - Additional inter-process communications

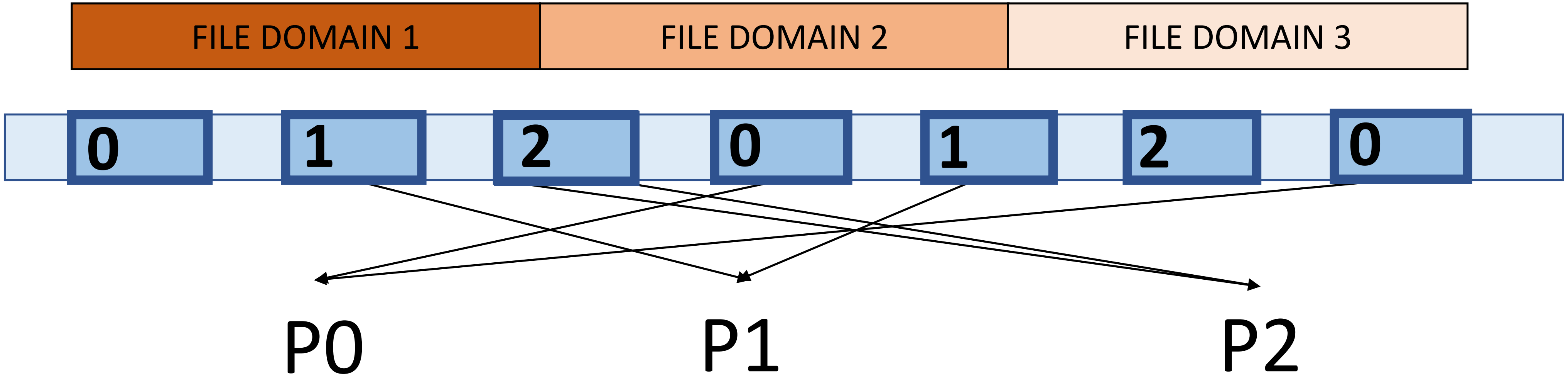
Two-phase I/O

- Phase 1
  - Processes analyze their own I/O requests
  - Create list of offsets and list of lengths
  - Everyone broadcasts start offset and end offset to others
  - Each process reads its own *file domain*

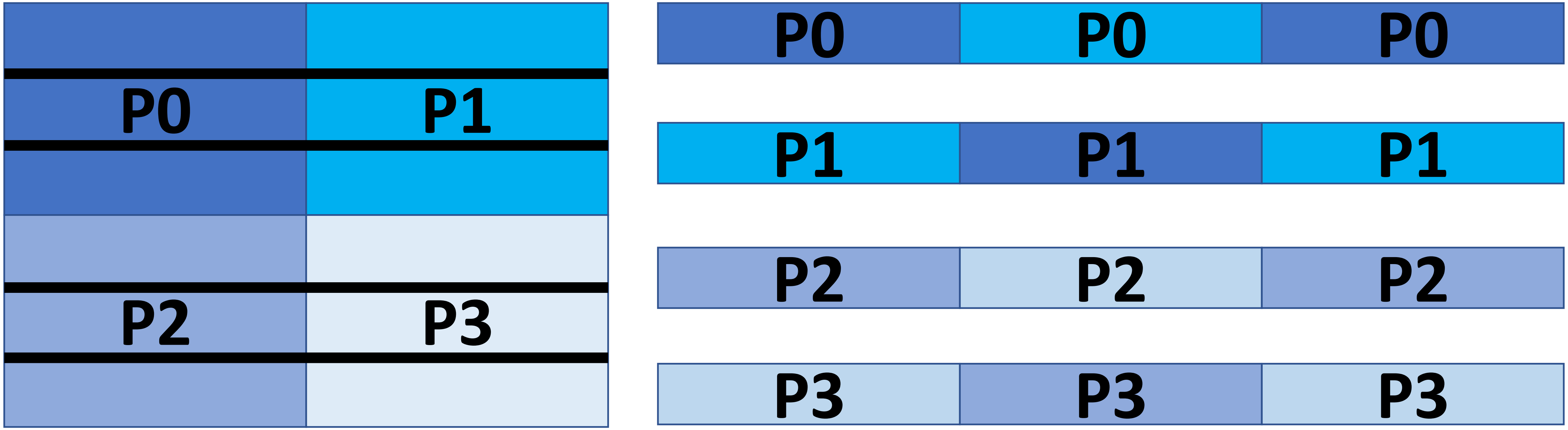


Two-phase I/O

- Phase 2
  - Processes analyze the file domains
  - Processes exchange data with the corresponding process

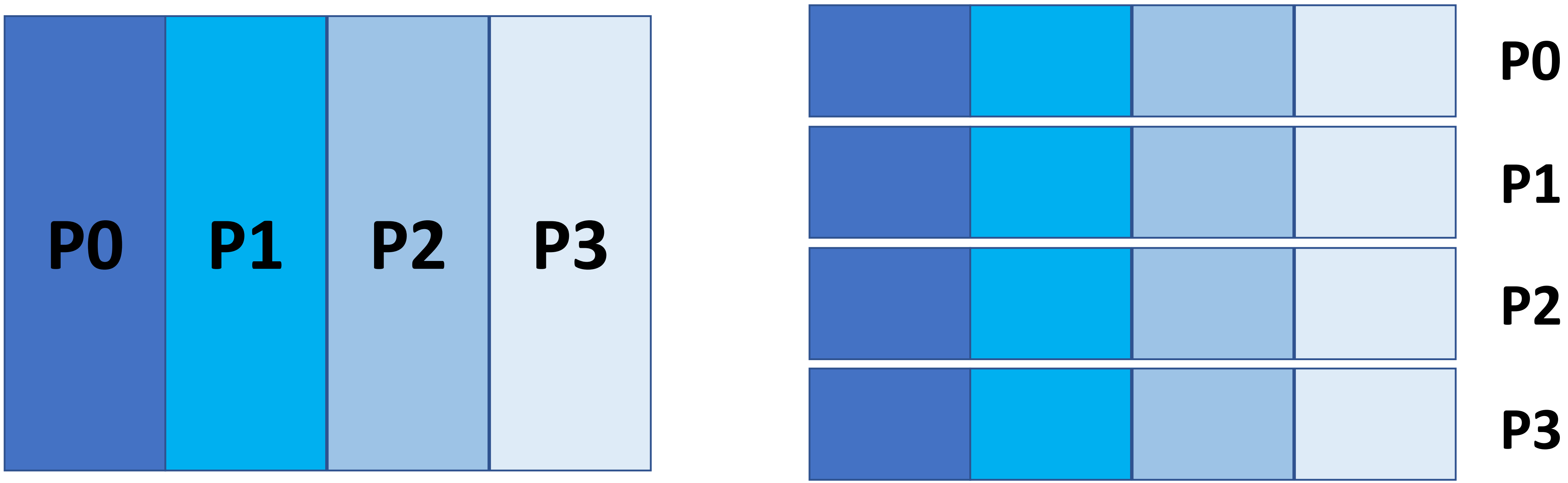


File domain – Example



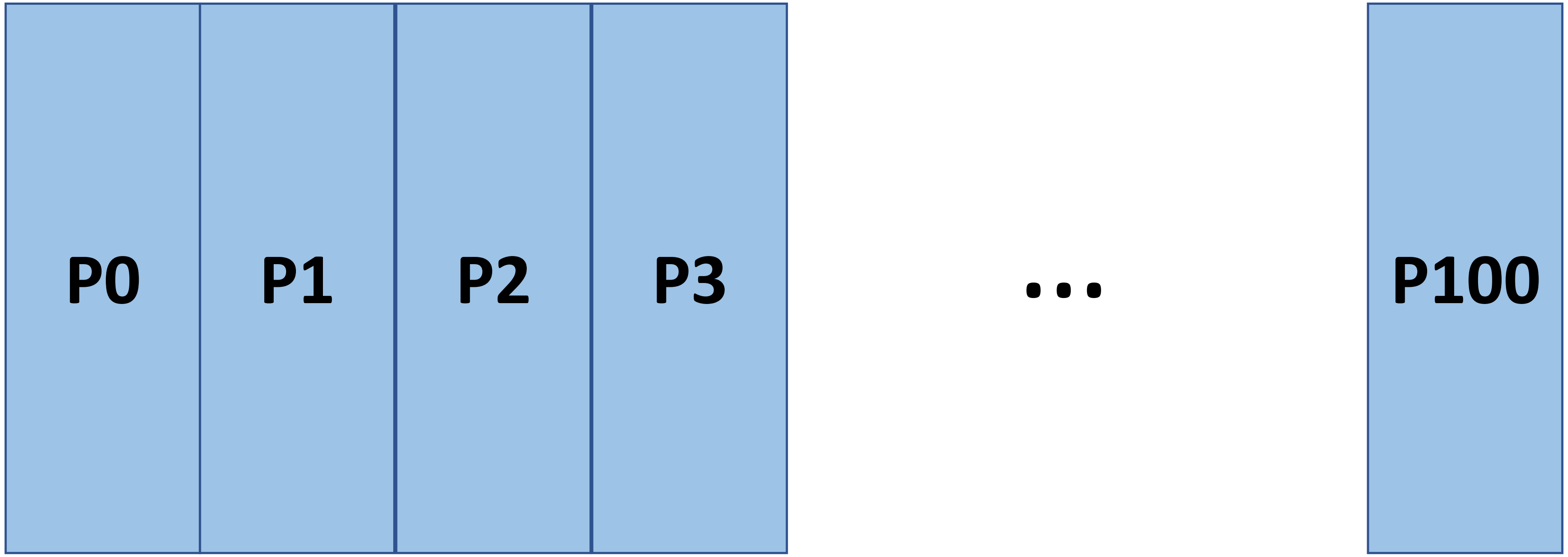
P0 and P1 exchange, P2 and P3 exchange

File domain – Example



Everyone needs data from every other process

Collective I/O



Communication may become bottleneck?