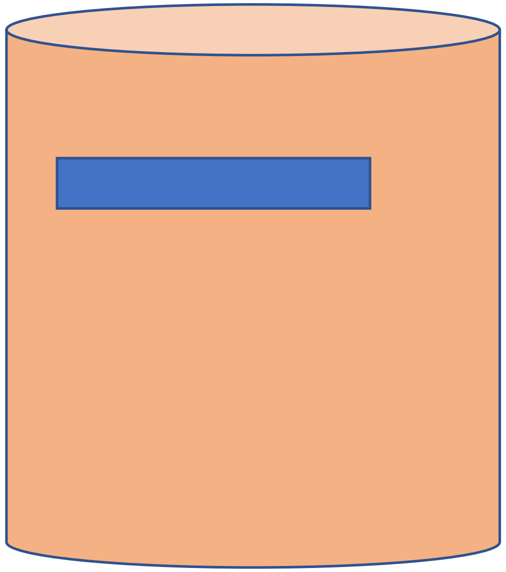
Parallel I/O - I

Apr 6, 2021

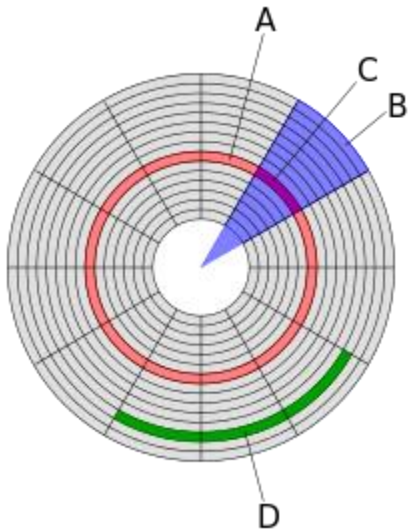
Sequential File Handling

```
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



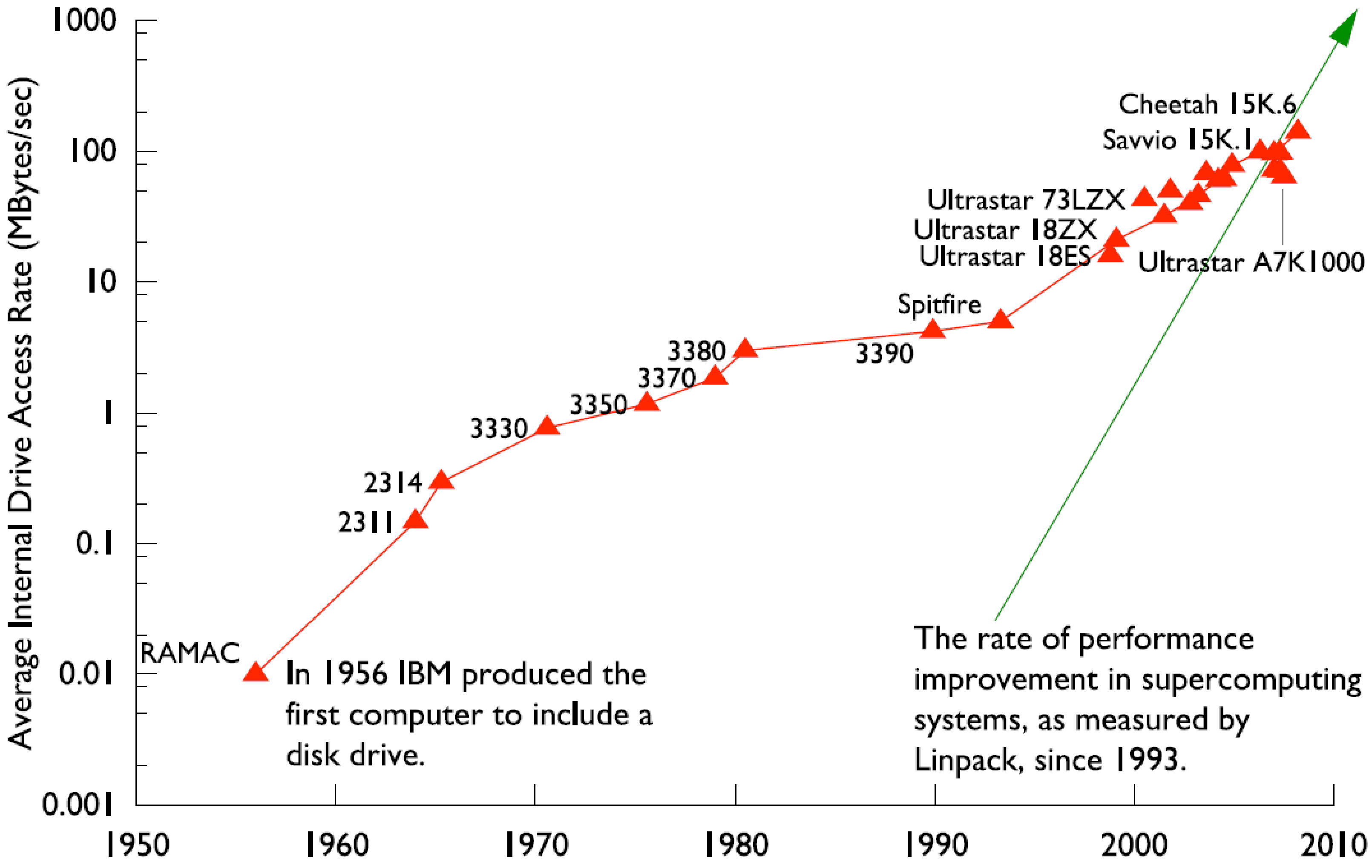
A. Track
B. Geometrical sector
C. Track sector
D. Cluster

[Source: Wikipedia]

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

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

Disk Access Rates



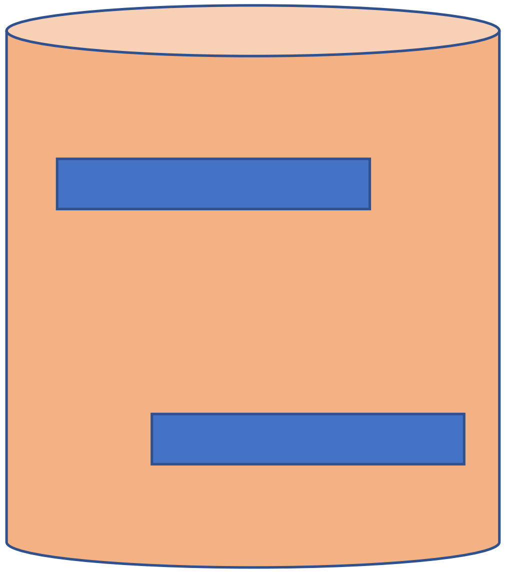
[Credit: R. Ross, ANL and R. Freitas, IBM]

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
```

Read bandwidth

```
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]

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

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

```c
#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;

}
~
```

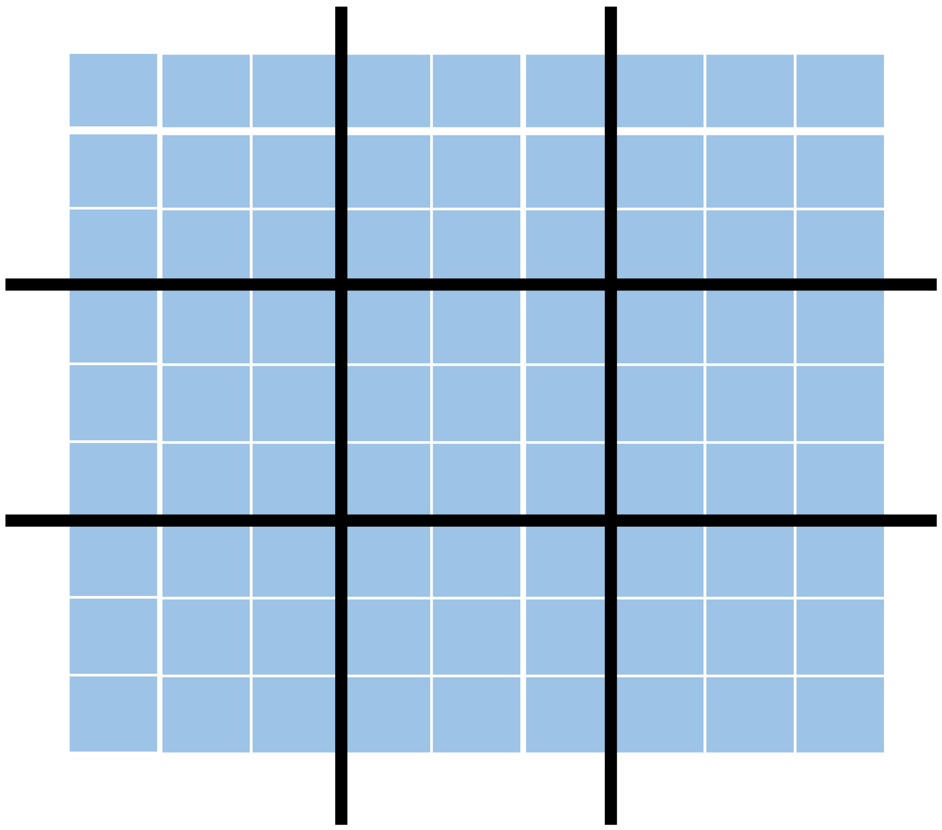
Independent writes

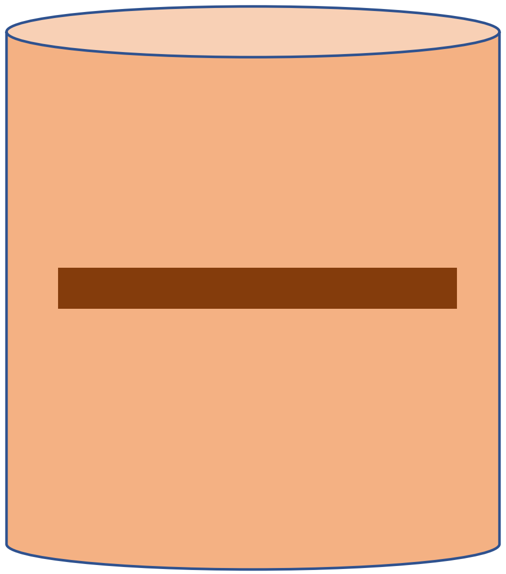
Writing to Different Files

```
class for j in 10000 100000 1000000 10000000 100000000 ; do time mpirun -np 1 ./1.sfile ${j} ; done

real    0m0.019s
user    0m0.008s
sys     0m0.001s

real    0m0.054s
user    0m0.017s
sys     0m0.000s

real    0m0.225s
user    0m0.100s
sys     0m0.002s

real    0m1.911s
user    0m0.849s
sys     0m0.095s

real    0m23.779s
user    0m11.918s
sys     0m0.559s
class
```

```
class for j in 10000 100000 1000000 10000000 100000000 ; do time mpirun -np 4 ./1.sfile ${j} ; done

real    0m0.483s
user    0m0.887s
sys     0m0.141s

real    0m0.144s
user    0m0.258s
sys     0m0.000s

real    0m0.841s
user    0m0.827s
sys     0m0.045s

real    0m4.251s
user    0m4.487s
sys     0m0.208s

real    0m46.416s
user    0m48.527s
sys     0m2.341s
```

Large Domain

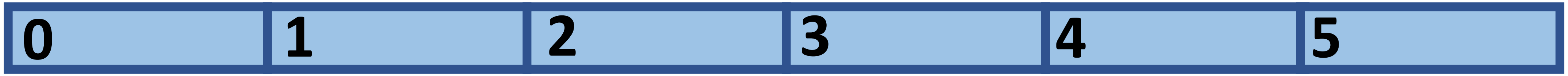




Simple Parallel I/O Code

```
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)
```

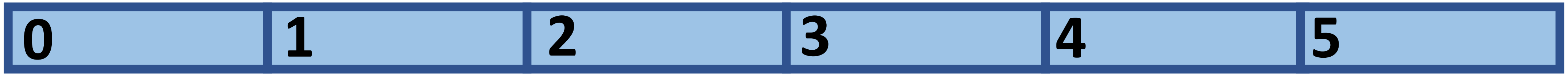
0	1	2	3	4	5

Explicit Offset

```
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]);
```

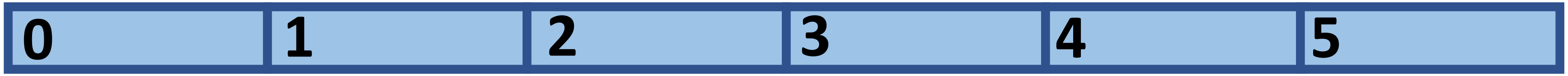
Set File View

```c
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]);
}
```

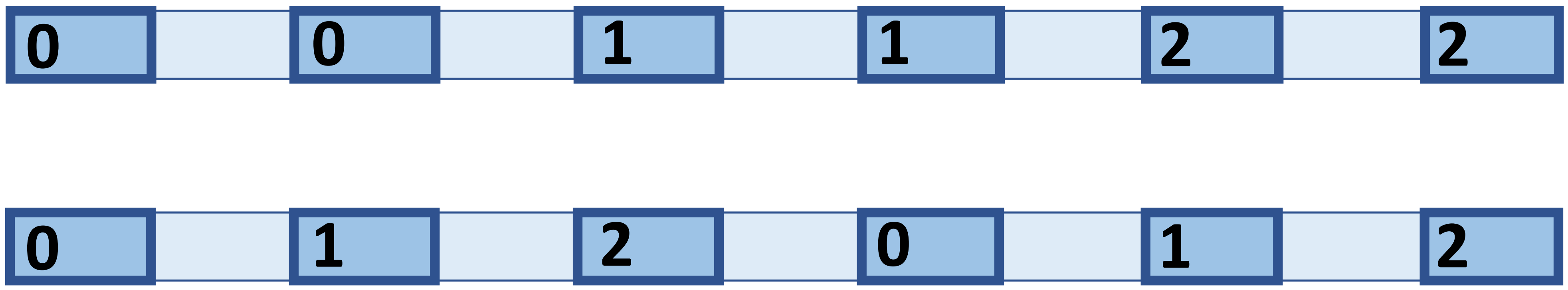


Features

- 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



Non-contiguous Accesses

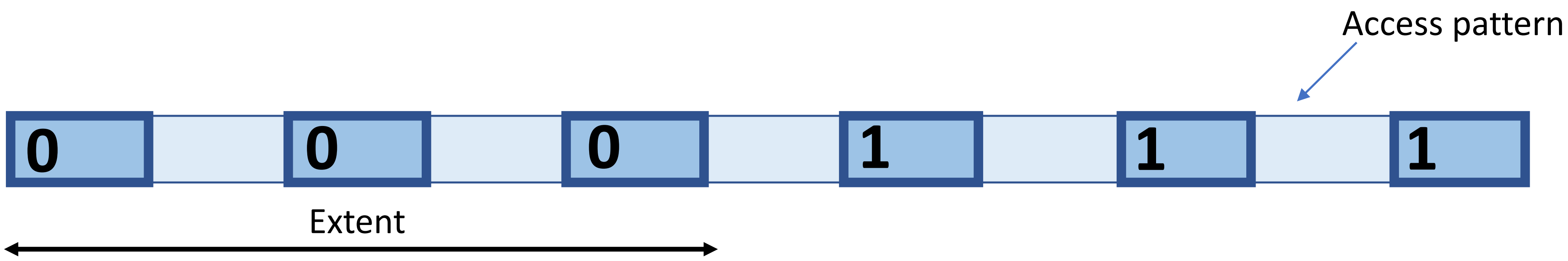


Frequently occurring file access pattern (non-contiguous)

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)

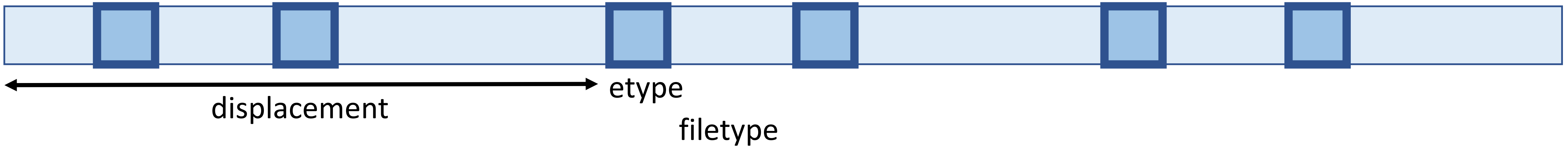
MPI_File_read_at (fh, offset2, buffer2, count2, MPI_INT, status)

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

Can we instead use one read call?

File View

Non-contiguous access pattern can be specified using a view



Each process can specify a view

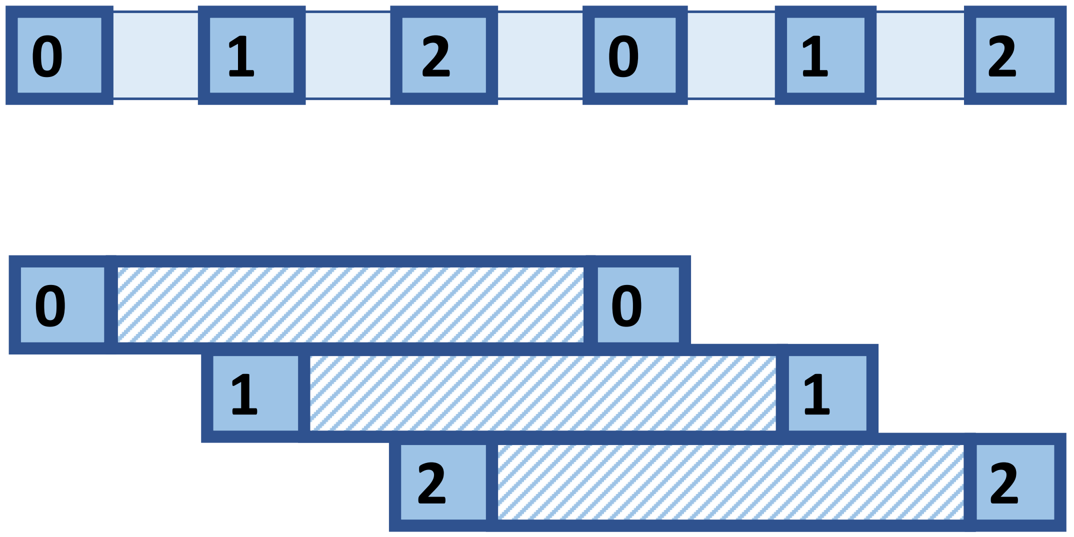
$$\left.\begin{array}{l}\text{displacement}\\\text{etype}\\\text{filetype}\end{array}\right\}\ \text{File view}$$

MPI_File_set_view (fh, disp, etype, filetype, “native”, MPI_INFO_NULL)

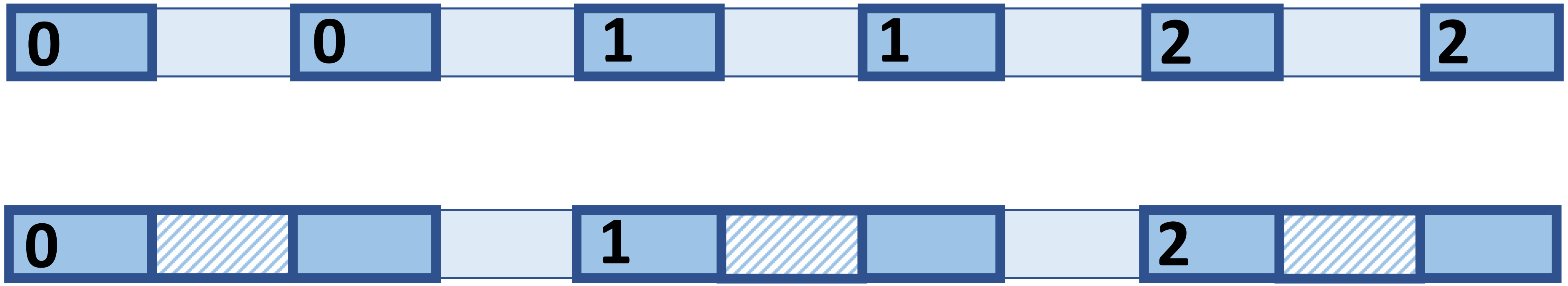
MPI_File_read (fh, buffer, count, MPI_INT, status)

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

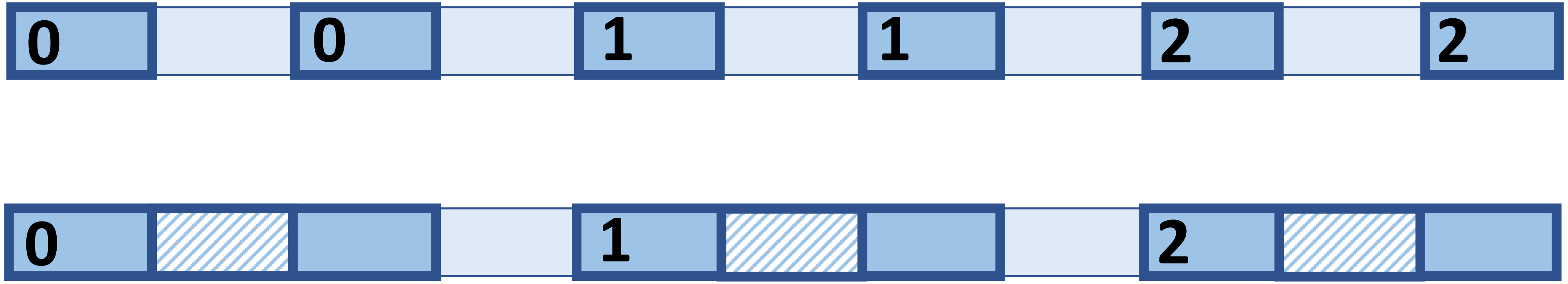


Optimization – Data Sieving



- 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