Parallel I/O - I

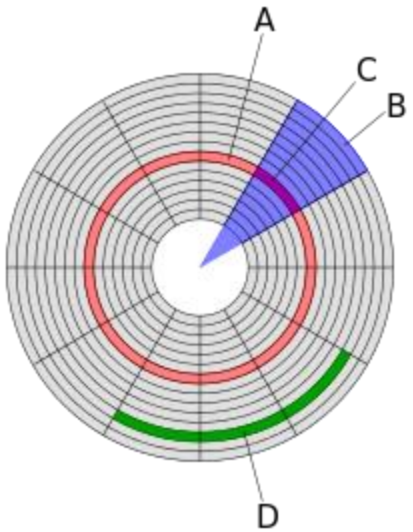
Feb 5, 2019

Sequential File Handling

```
char instruction[] = "Start your projects"
FILE *fp = fopen ("/scratch/smallfile", "w")
fwrite (instruction,  sizeof(char), sizeof(instruction), fp)
fclose (fp)
```

Sequential I/O

- 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

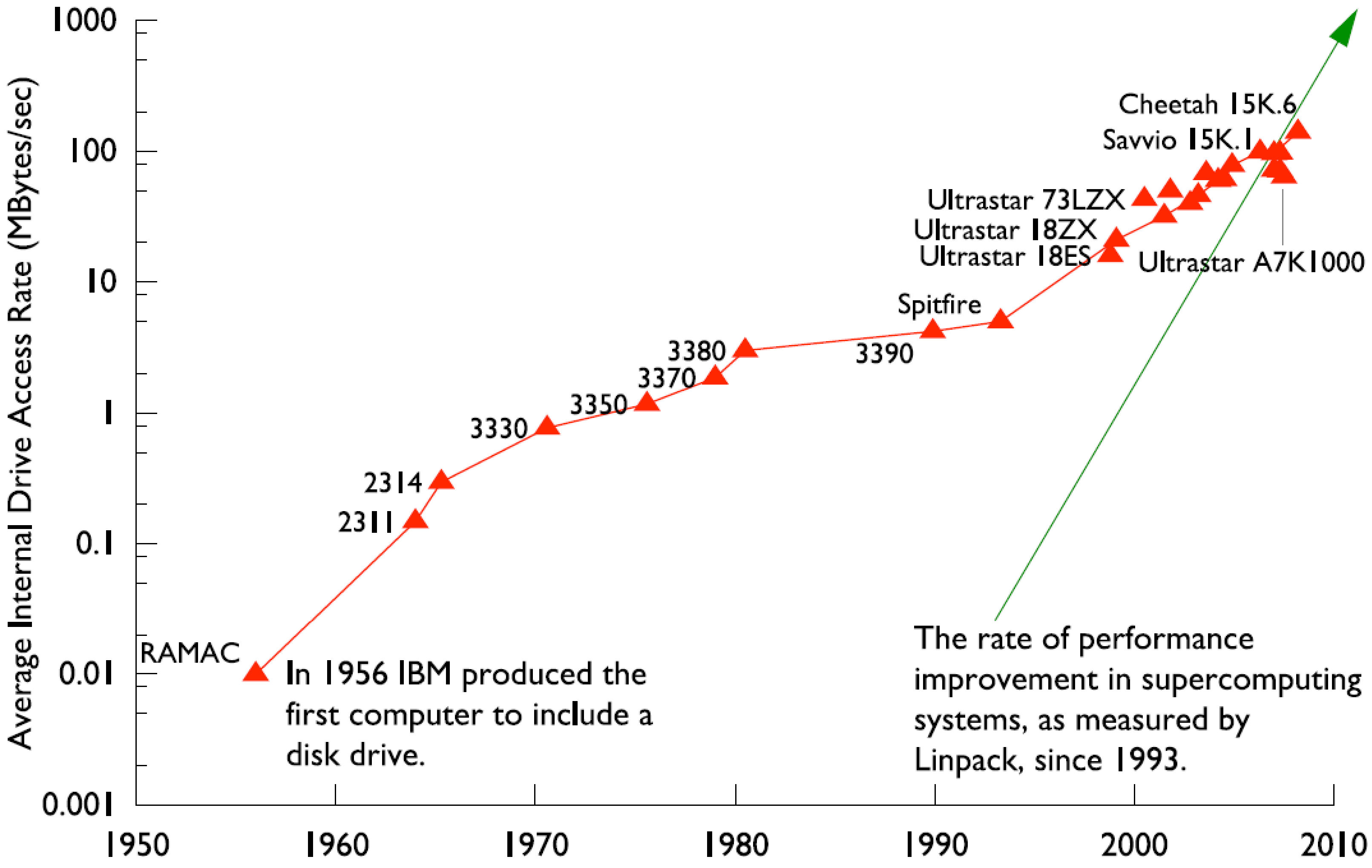


A. Track
B. Geometrical sector
C. Track sector
D. Cluster

[Source: Wikipedia]

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

Disk Access Rates



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

Data Requirements

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

[Source: 2008 report, S. Klasky]

Parallel I/O

What?

- Every process reads and writes files in parallel
- Simultaneous access to storage

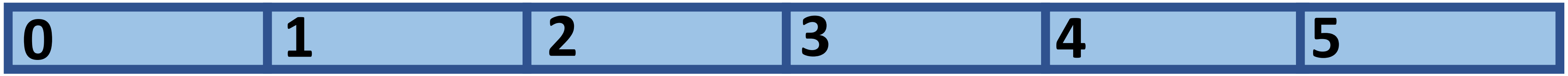
Why?

- 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

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

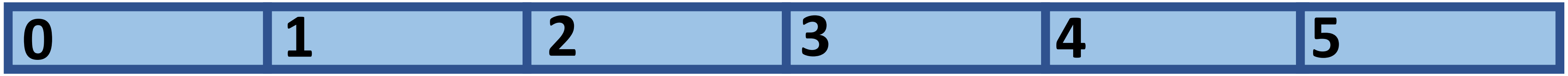


Observations

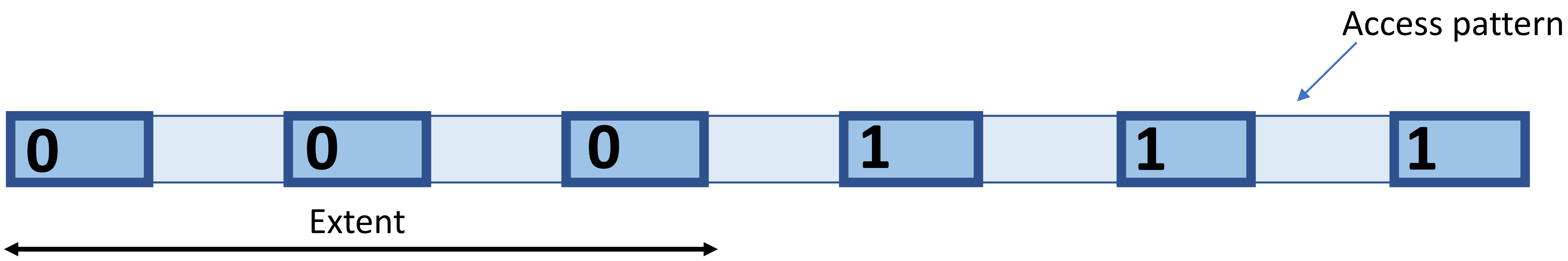
- Multiple processes access a common file
- Multiple processes access the same file at the same time
- Multiple seeks issued at the same time
- Each process reads a contiguous chunk
- Individual file pointers

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)
```



Multiple 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

displacement
etype
filetype

File view

Default file view: When file is opened

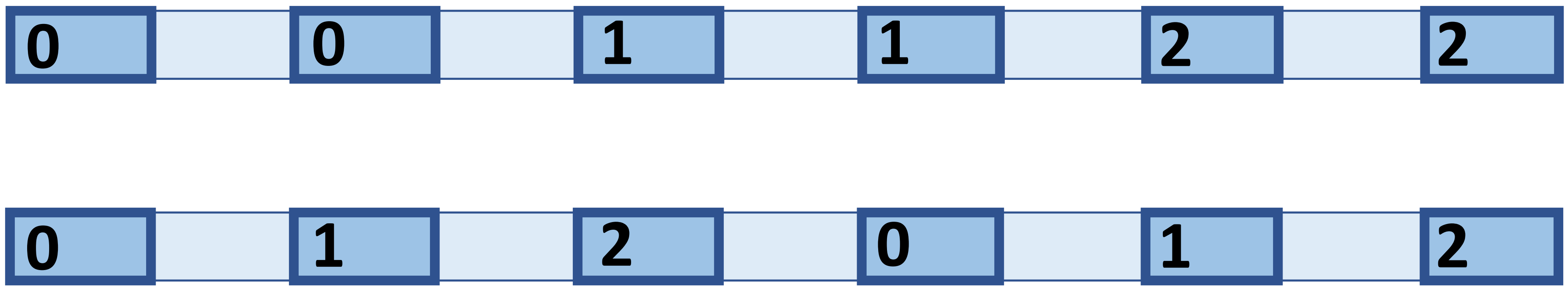
displacement=0

etype and filetype are MPI_BYTE

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

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

Non-contiguous Accesses

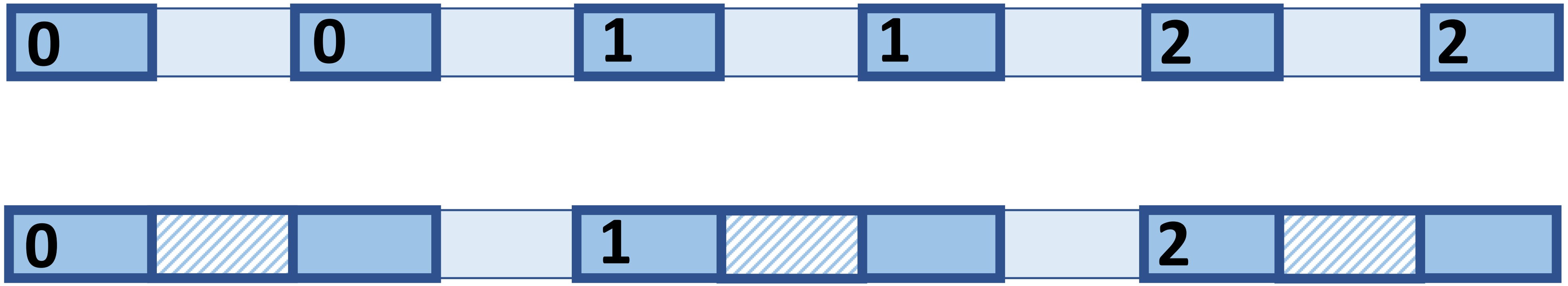


Frequently occurring file access pattern (non-contiguity)

What is the problem with non-contiguous accesses?

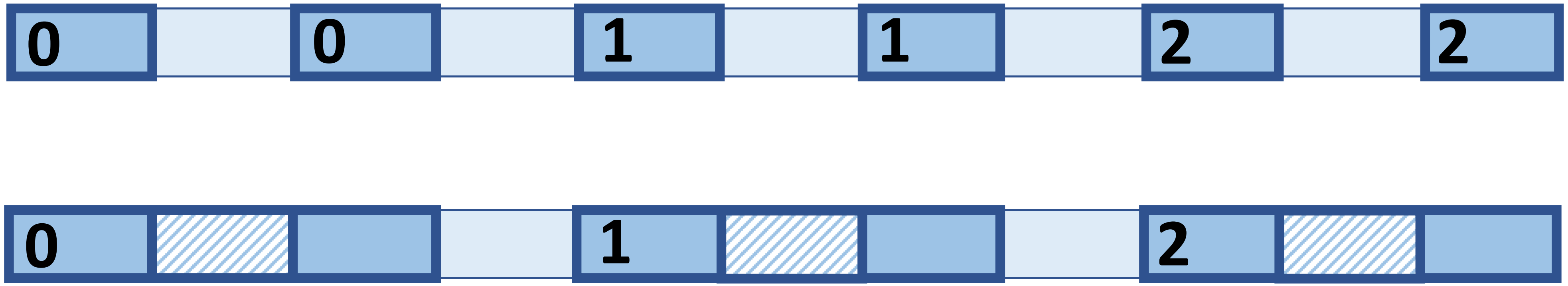
Programmability resolved, but what about performance?

Optimization I – Data Sieving



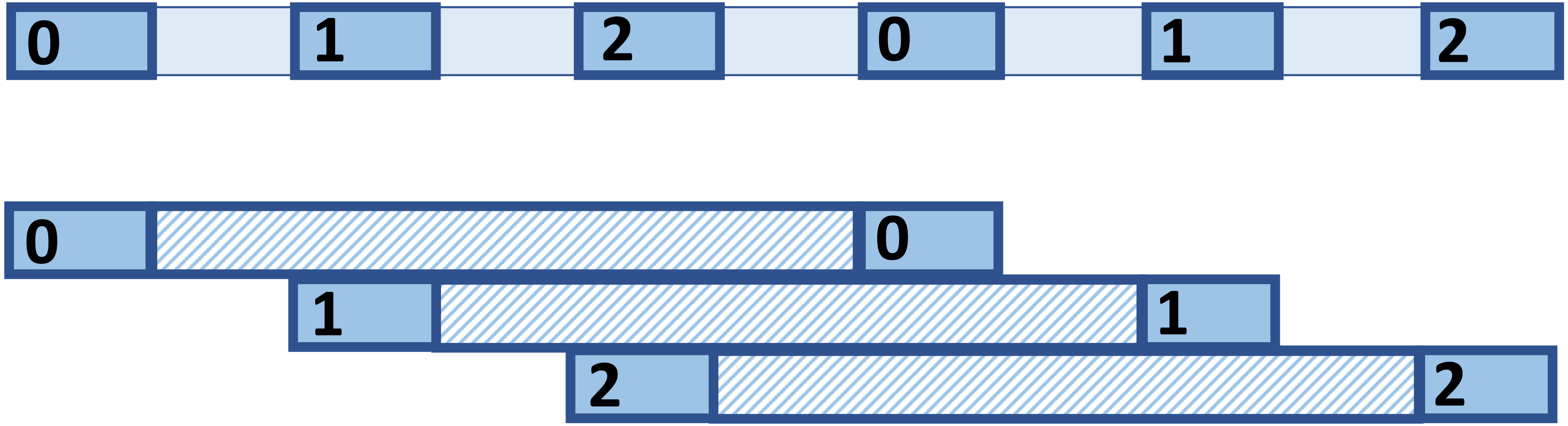
- Make large I/O requests and extract the data that is really needed
- Huge benefit of reading large, contiguous chunks

Data Sieving for Writes



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

Data Sieving – Interleaved data



Q: What is the problem here?

Solution – Lock the relevant portions in the file

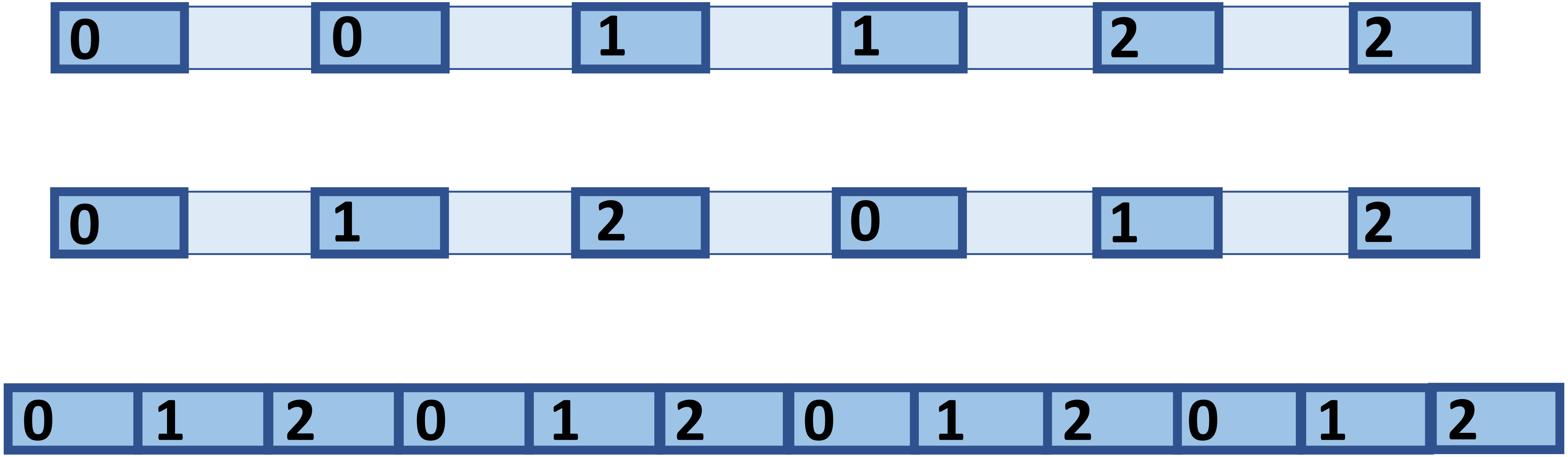
User-controlled MPI-IO Parameters

- 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

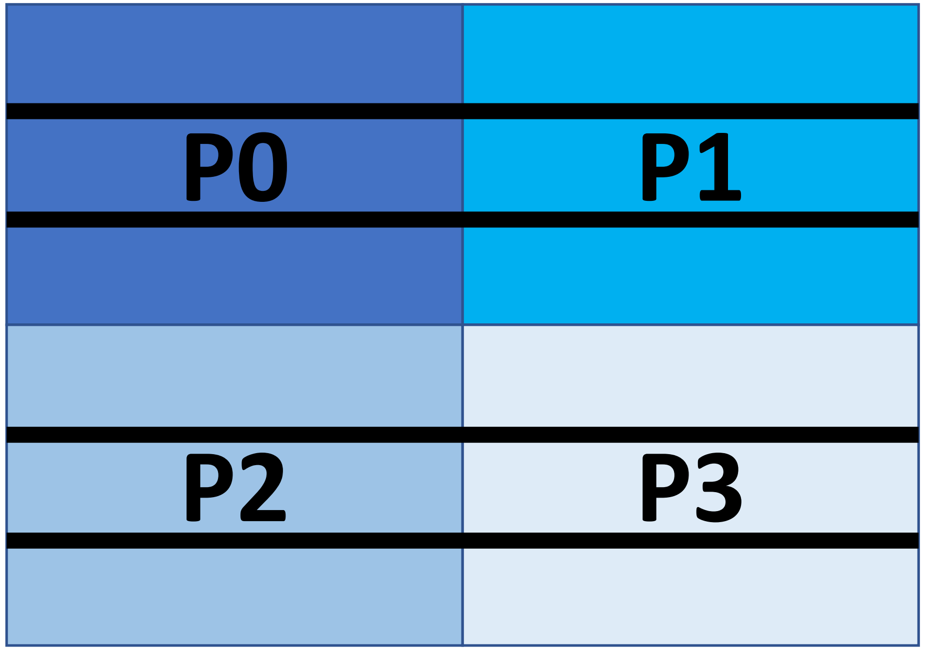
Parallel I/O Classification

- Independent I/O (we saw till now)
- 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?

Optimization II – Collective I/O

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

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

MPI_File_close (&fh)

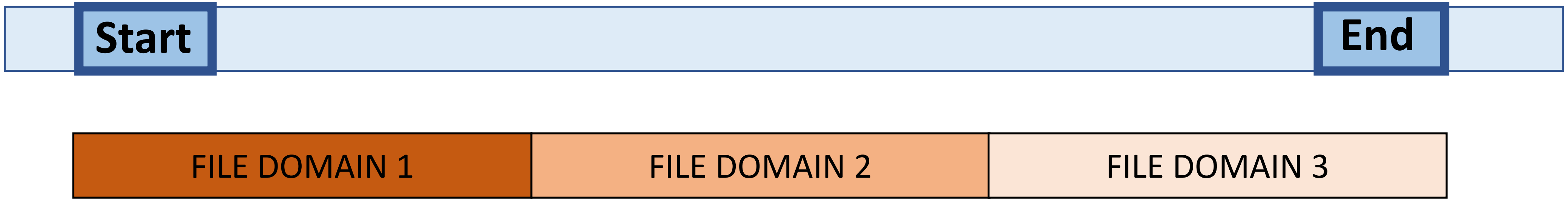
Two-phase I/O

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

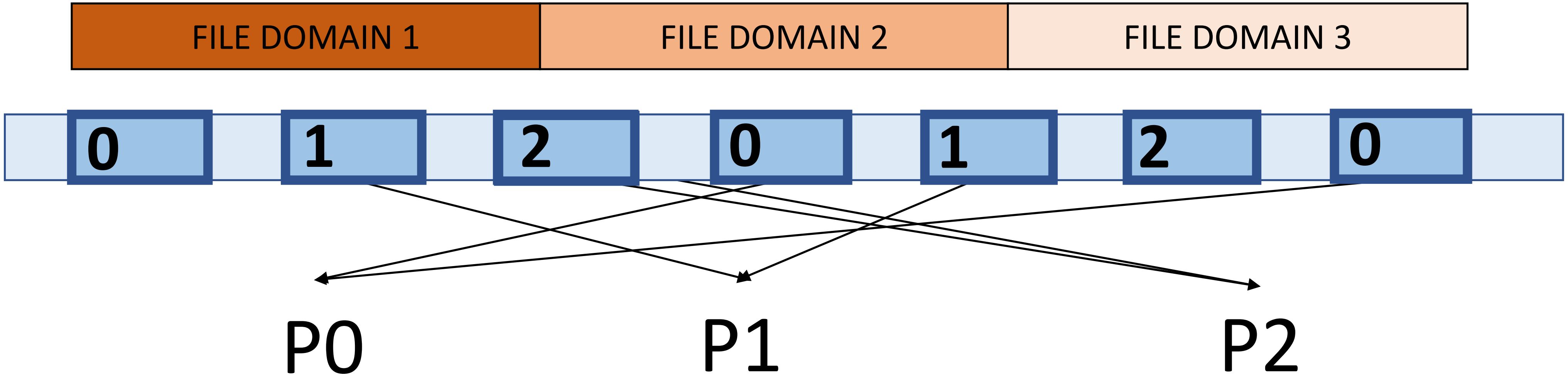
Two-phase I/O

- 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*

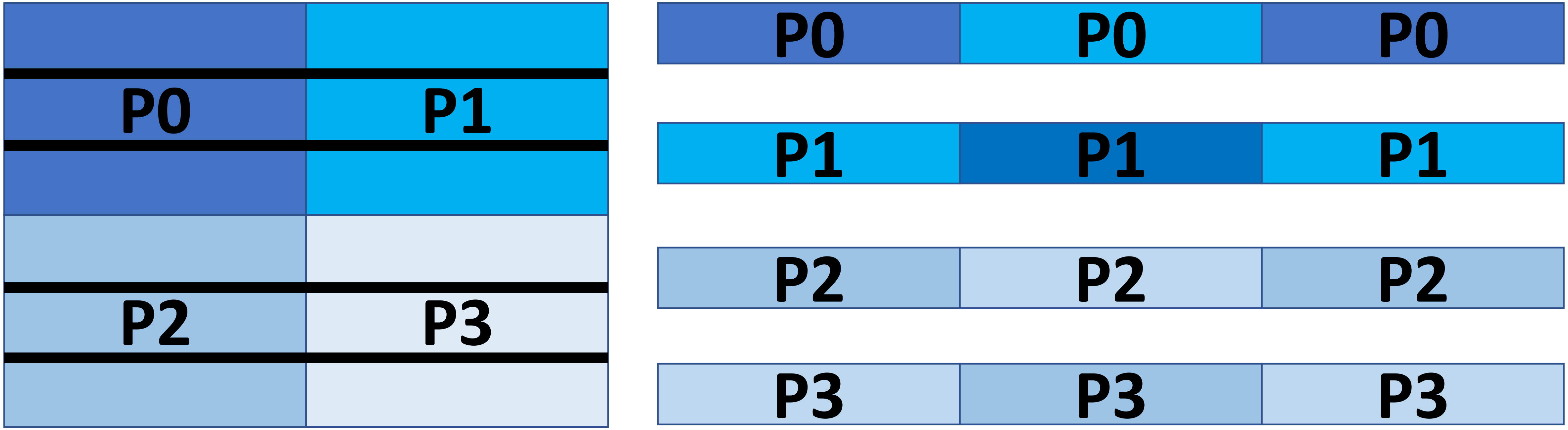


Two-phase I/O

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



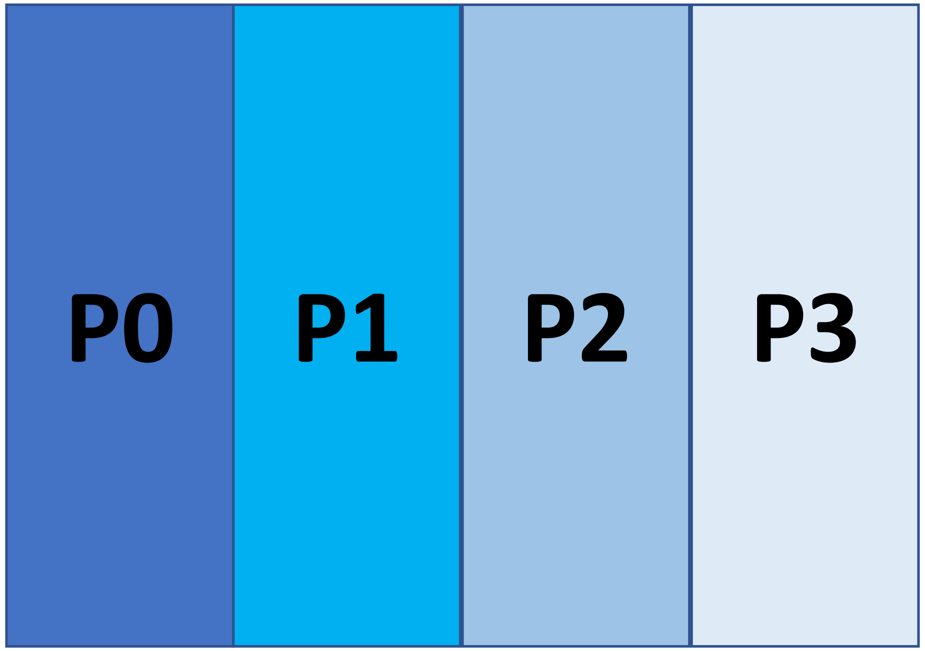
File domain – Example

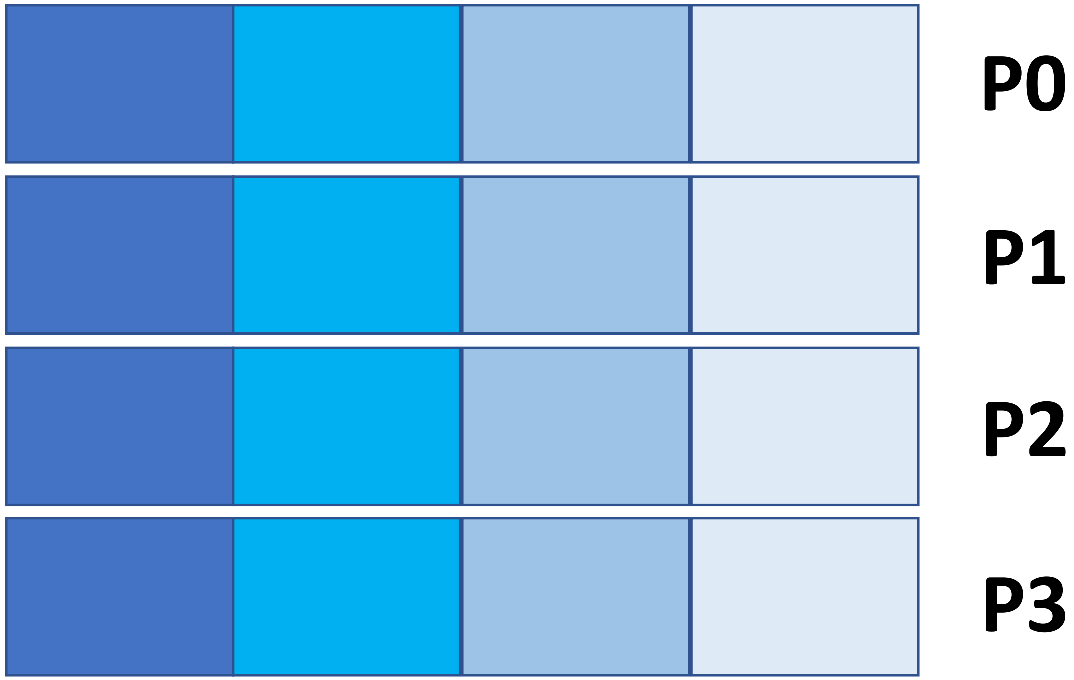


P0 and P1 exchange, P2 and P3 exchange

How should they communicate?

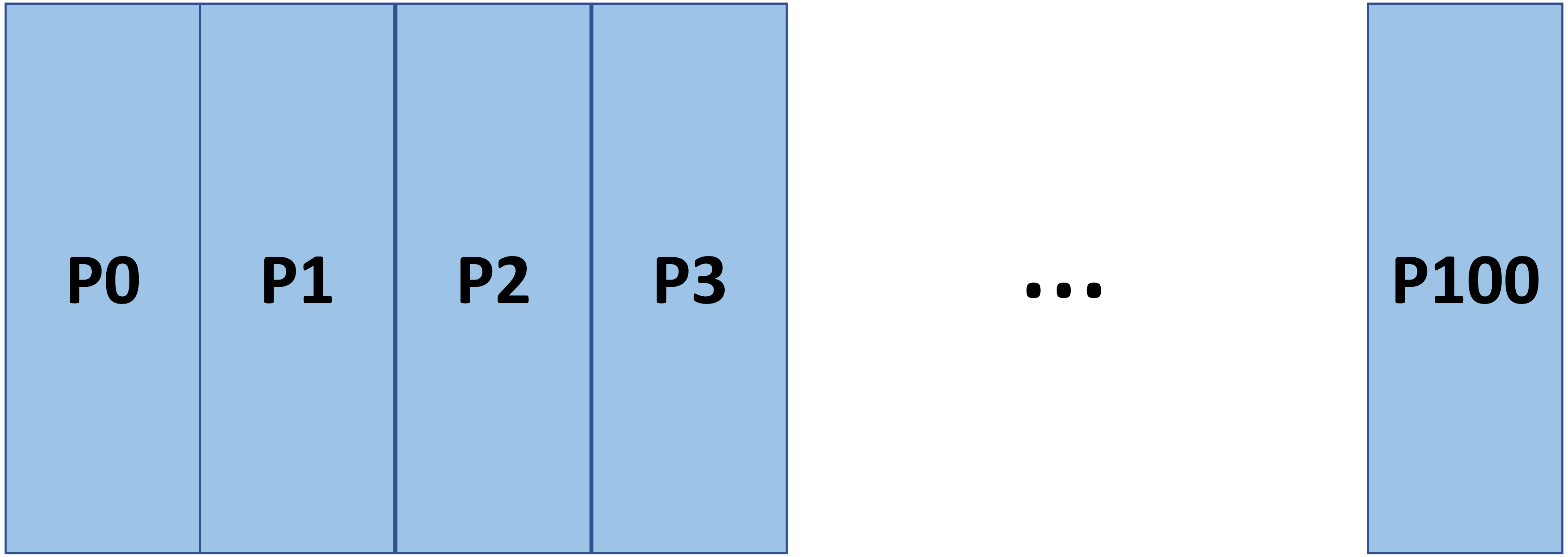
File domain – Example





Everyone needs data from every other process
How should they communicate?

File domain – Example



Communication may become bottleneck, as well as file I/O?

User-controlled Parameters

- Number of aggregators
- Buffer size in aggregators