Parallel I/O 101

Ivan Girotto – igirotto@ictp.it
International Centre for Theoretical Physics (ICTP)

Text Files Vs Binary Files

- Text files are accessed sequentially
  - A file pointer keeps tracks of where was the last read/write made from and it makes the next read/write from the right next location
- Binary files are accessed at any point
  - Posix Linux *fseek(…)* teels di O.S. where to place the point to the next read/write. All following reads/write operation are going to be sequential if not specified otherwise by an other *fseek* call
- How big is a binary file containing one single integer number?
- How big is a file

Why is Parallel I/O difficult?

- Cannot have multiple processes writing a file
  - Unix cannot cope with this
  - data cached in units of disk blocks (eg 4K) and is *not coherent*
  - not even sufficient to have processes writing to distinct parts of file
- Even reading can be difficult
  - 1024 processes opening a file can overload the filesystem (fs)
- Data is distributed across different processes
  - processes do not in general own contiguous chunks of the file
  - cannot easily do linear writes
  - local data may have halos to be stripped off
- Parallel file systems may allow multiple access
  - but complicated and difficult for the user to manage

Why is Parallel I/O difficult?

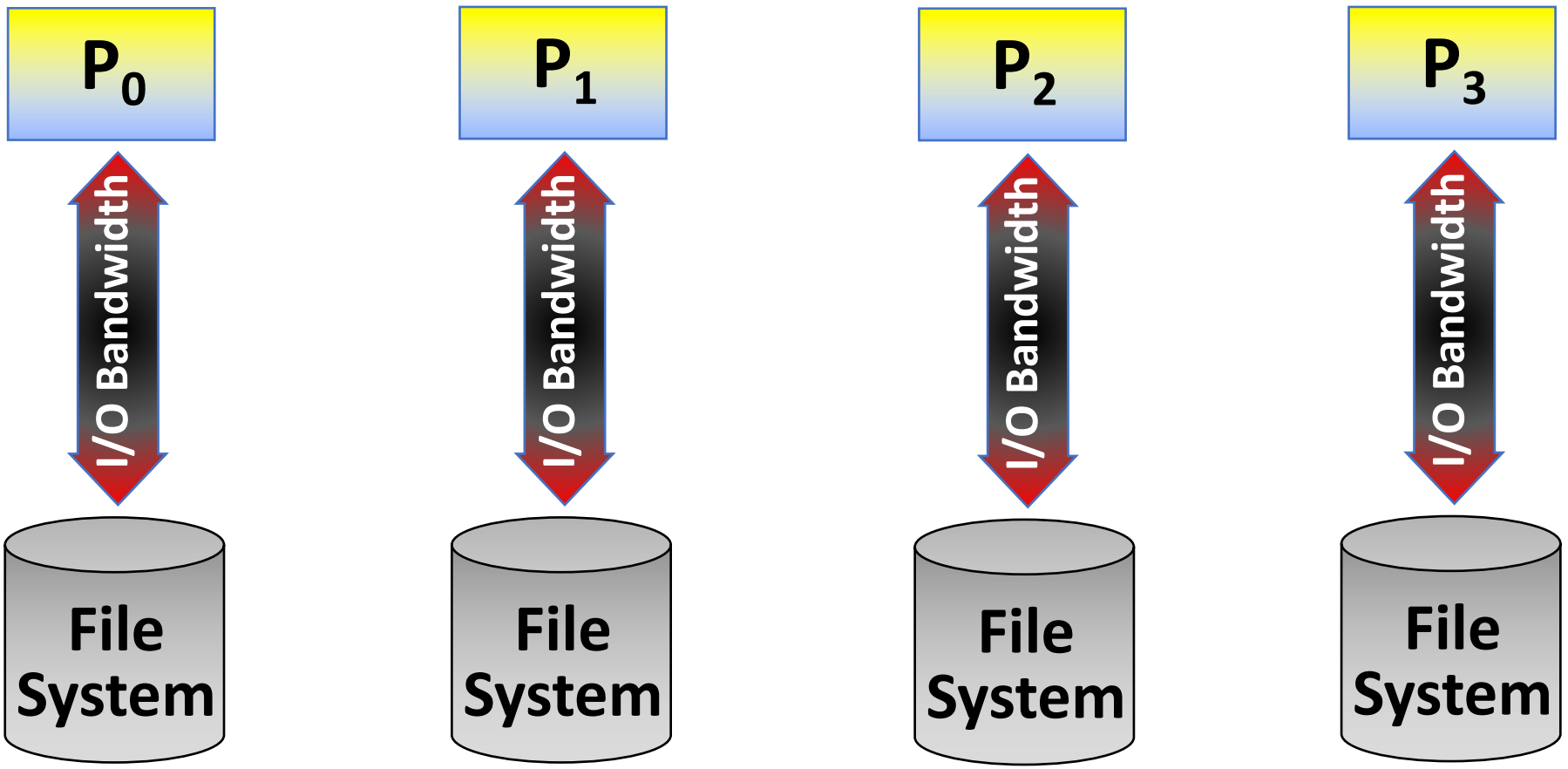
- Easy to solve in shared memory
  - imagine a shared array called **`data`**

```
begin serial region
      open the file
      write data to the file
      close the file
end serial region
```

- Simple as every thread can access shared data
  - may not be efficient but it works
- But what about message-passing?

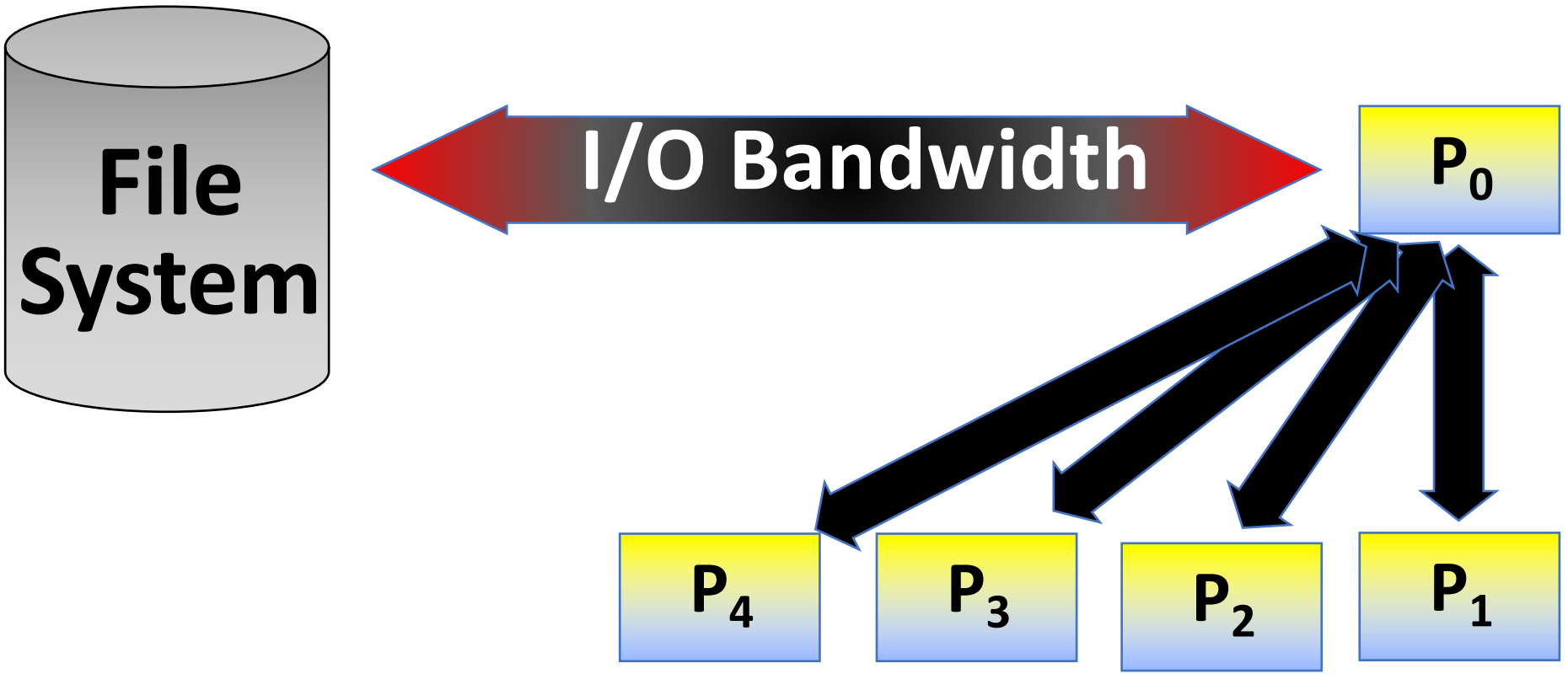
Parallel I/O simplified version

- All processors write their own files
- Normally requires a non-trivial post-processing effort



Parallel I/O sequential version

- All processors send/receive data to/from the master process
- Not the most efficient but portable and effective



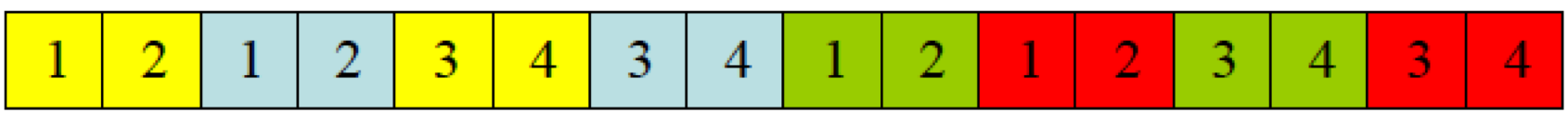
Parallel I/O sequential version

```
if( !rank ){

      print_matrix( mat, size_loc );
      int count;
      for( pe = 1; pe < npes; pe++ ){
            MPI_Recv( mat, SIZE * size_loc, MPI_DOUBLE,
                  pe, pe, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            if( rest && count >= rest )
                  print_matrix( mat, size_loc - 1);

            else print_matrix( mat, size_loc );
      }
    }
else
      MPI_Send( mat, SIZE * size_loc, MPI_DOUBLE,
                  0, rank, MPI_COMM_WORLD );
```

Why is Distributed Parallel I/O difficult?

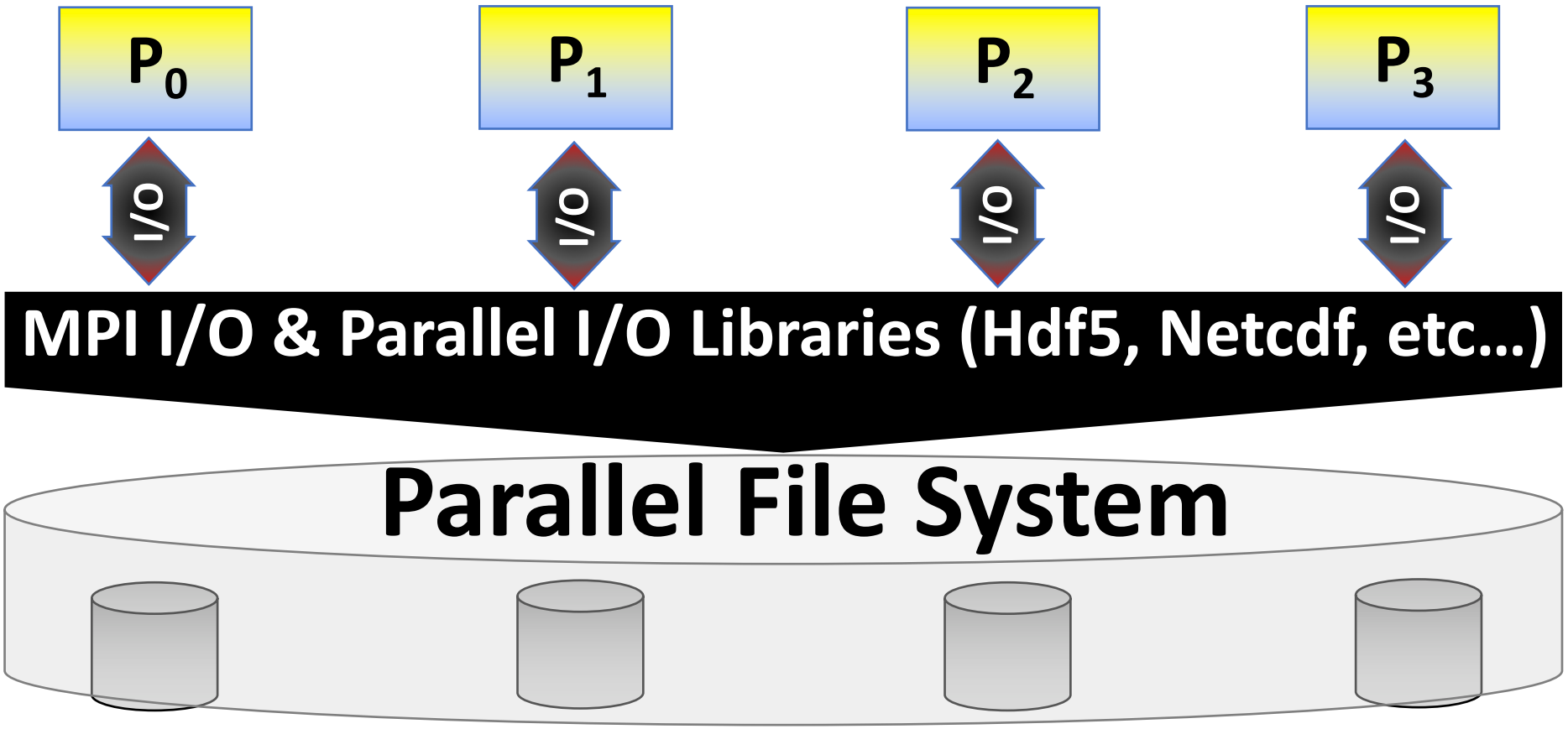
Parallel Data

File



For non-contiguous data is better to use MPI_Datatype!!

Parallel I/O



The parallel file system handles concurrent access to the file from multiple processes

Load Balance

- I/O is quite efficient on local disks, it is terrible on parallel systems: order of magnitude less efficient
- How often do I write on disk?
- Impact the application
  - Scaling
  - Time to solution
- Coarse grain approach: reduce the domain size by saving only one every n points, averaging among the points
  - Not always applicable
  - Good for intermediated steps

Conventional Data Format in Scientific Computing

- Several scientific community have defined a common way to store data: SEG-Y, HDF5, NETCDF
- The idea is to to create a common metadata description. The metadata is used to describe the content of the file
  - how many types of entries, how many entries per each type, how is the geometry of the data saved in the file -> 1D, 2D, 3D etc…
  - Allow quick access to the file
  - Compressed format
  - Extremely convenient for visualization -> the viz tool know the formats!
  - Good for potability