# I/O in Parallel 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 text file containing one single integer number?

# Binary Files: portability issue

## Data portability: number representation

There are two different representations:

**Little Endian**

Byte3 Byte2 Byte1 Byte0

Alpha, PC (Windows/Linux)

will be arranged in memory as follows:

Base Address+0 Byte0
Base Address+1 Byte1
Base Address+2 Byte2
Base Address+3 Byte3

**Big Endian**

Byte3 Byte2 Byte1 Byte0

will be arranged in memory as follows:

Base Address+0 Byte3
Base Address+1 Byte2
Base Address+2 Byte1
Base Address+3 Byte0

Unix (IBM, SGI, SUN…)

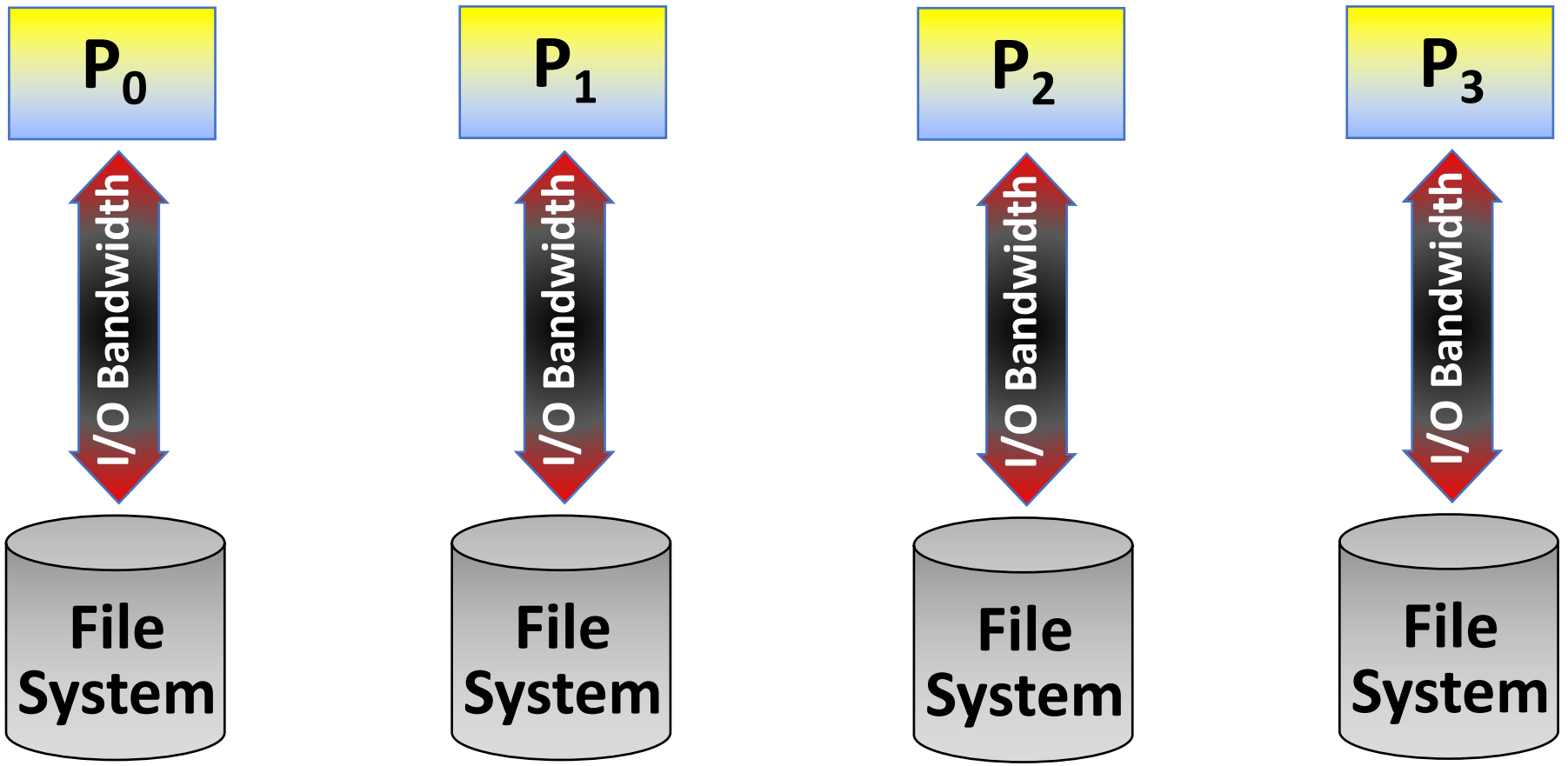
# Why is Parallel I/O difficult?

- 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
```

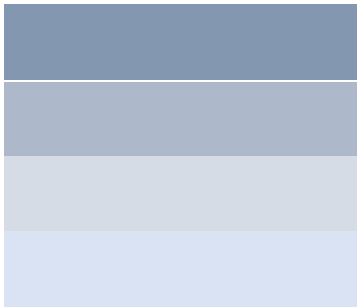
- Simple as every thread can access shared data
  - may not be efficient but it works
- But what about message-passing?

# SPMD/MPMD and I/O

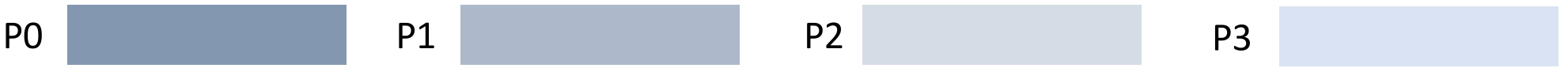


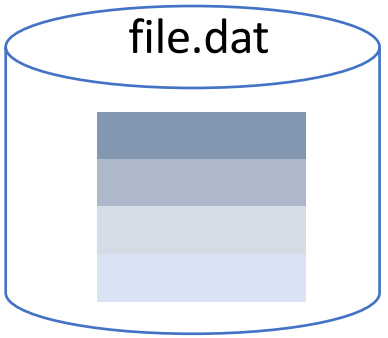

- **Normally all processes see the same file system, take advantage of it!**

# Distributed Parallel I/O



Logical
Rappresentation




Logical
Rappresentation

# Why is Parallel I/O difficult?

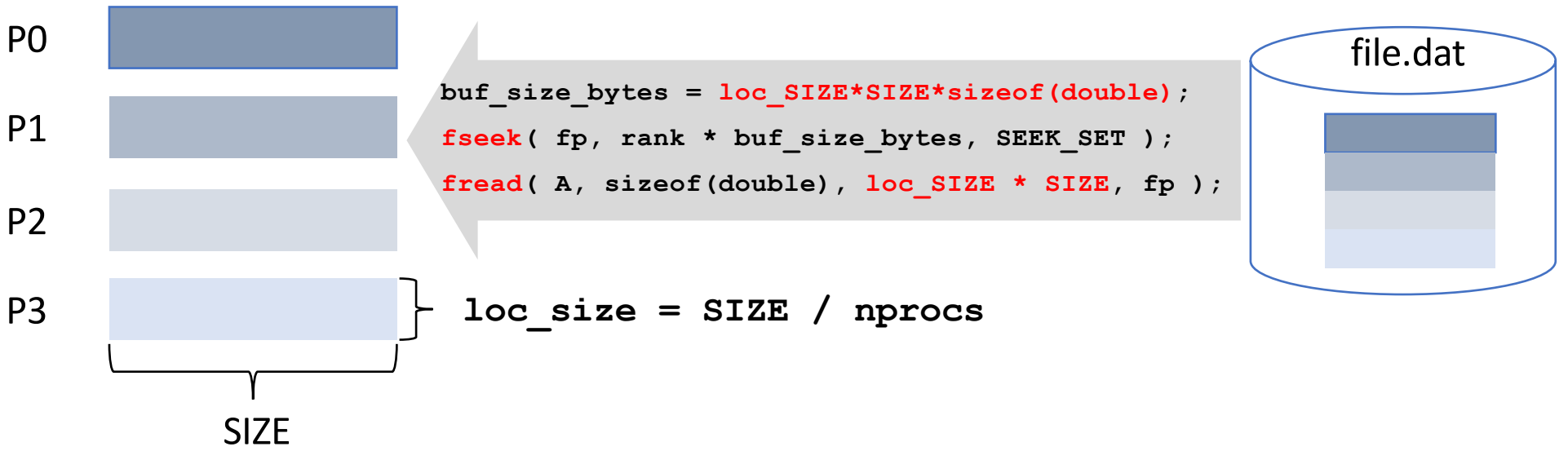
- Cannot have multiple processes writing  a file
  - Unix cannot cope with this
  - data cached in units of disk blocks (eg 4K) and is *not coherent*
  - not even sufficient to have processes writing to distinct parts of file
- Even reading can be difficult
  - 1024 processes opening a file can overload 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

# Access to files in parallel

- It requires a serialized access from one or more processes in case of a text file (sequential access)
- It can be performed in parallel in case of binary files (random access)
- Common sequence for creating and writing a binary file:

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

  double * A;
  int i = 0;
  FILE * fp;

  A = (double *) malloc( SIZE * SIZE * sizeof(double) );

  for( i = 0; i < SIZE * SIZE; i++ ){

    A[i] = (double) ( rand() % 1000 + 1 );
  }

  fp = fopen( "matrix.dat", "w" );
  fread( A, sizeof(double), SIZE * SIZE, fp );
  fclose( fp );

  free( A );

  return 0;
}
```

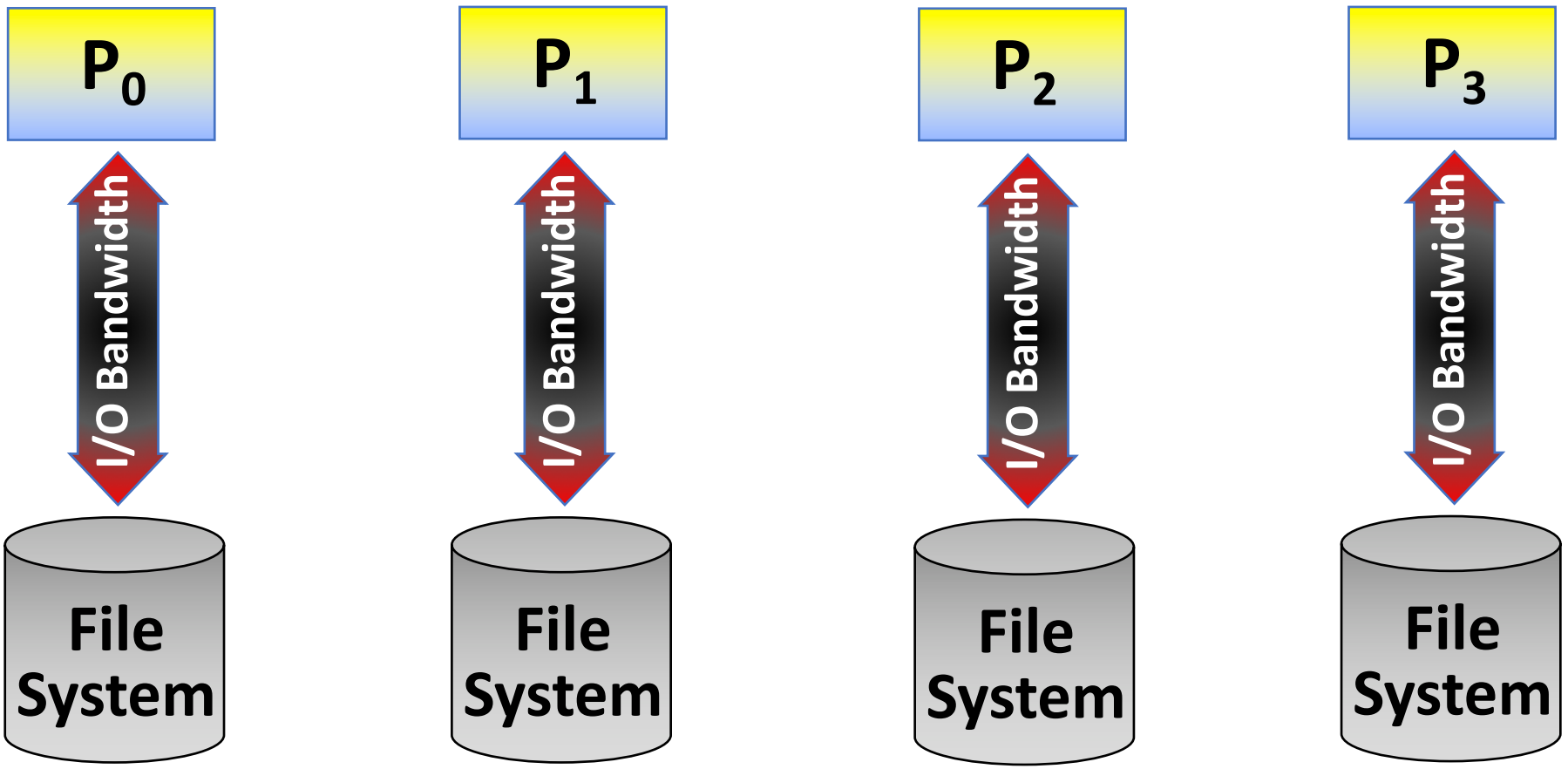
# How to read a binary file in parallel




The fseek() function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

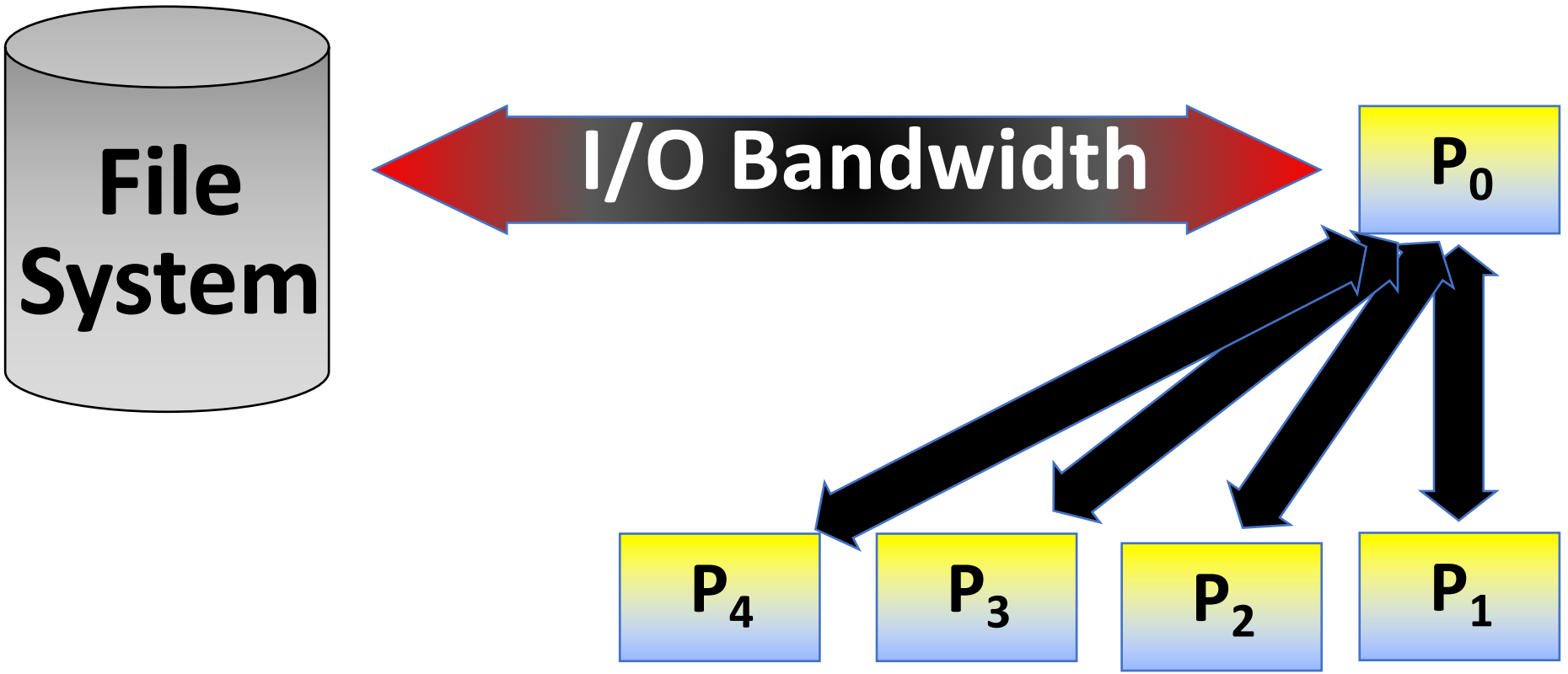
# Parallel I/O simplified version

- 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

```
void read_matrix( double * mat, int n_rows, int n_cols, FILE * fp ){

  int i, j;

  for( i = 0; i < n_rows; ++i ){
        for( j = 0; j < n_cols; ++j ){
            fscanf( fp, "%lg", &( mat[ i * n_cols + j ] )  );
        }
    }
}

void dist_matrix( double * mat, int n_rows, int n_cols, int rank, int npes, FILE * fp ){

  int count;
  double * buf_mat = (double *) malloc( n_rows * n_cols * sizeof(double) );

  if( rank == MPI_PROC_ROOT ){
    read_matrix( mat, n_rows, n_cols, fp );
    for( count = 1; count < npes; count++ ){
      read_matrix( buf_mat, n_rows, n_cols, fp );
      MPI_Send( buf_mat, n_rows * n_cols, MPI_DOUBLE, count, TAG, MPI_COMM_WORLD );
    }
  }
  else
    MPI_Recv( mat, n_rows * n_cols, MPI_DOUBLE, MPI_PROC_ROOT, TAG,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );

  free( buf_mat );
}
```

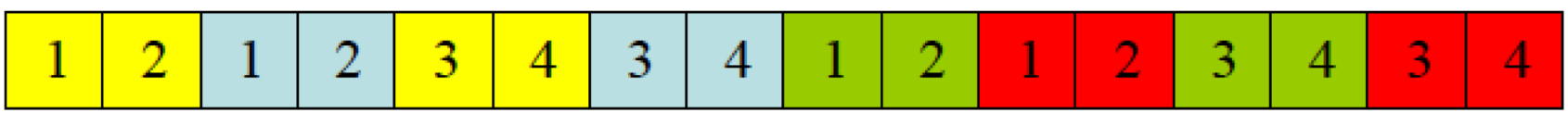
# Parallel I/O sequential version

```
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?

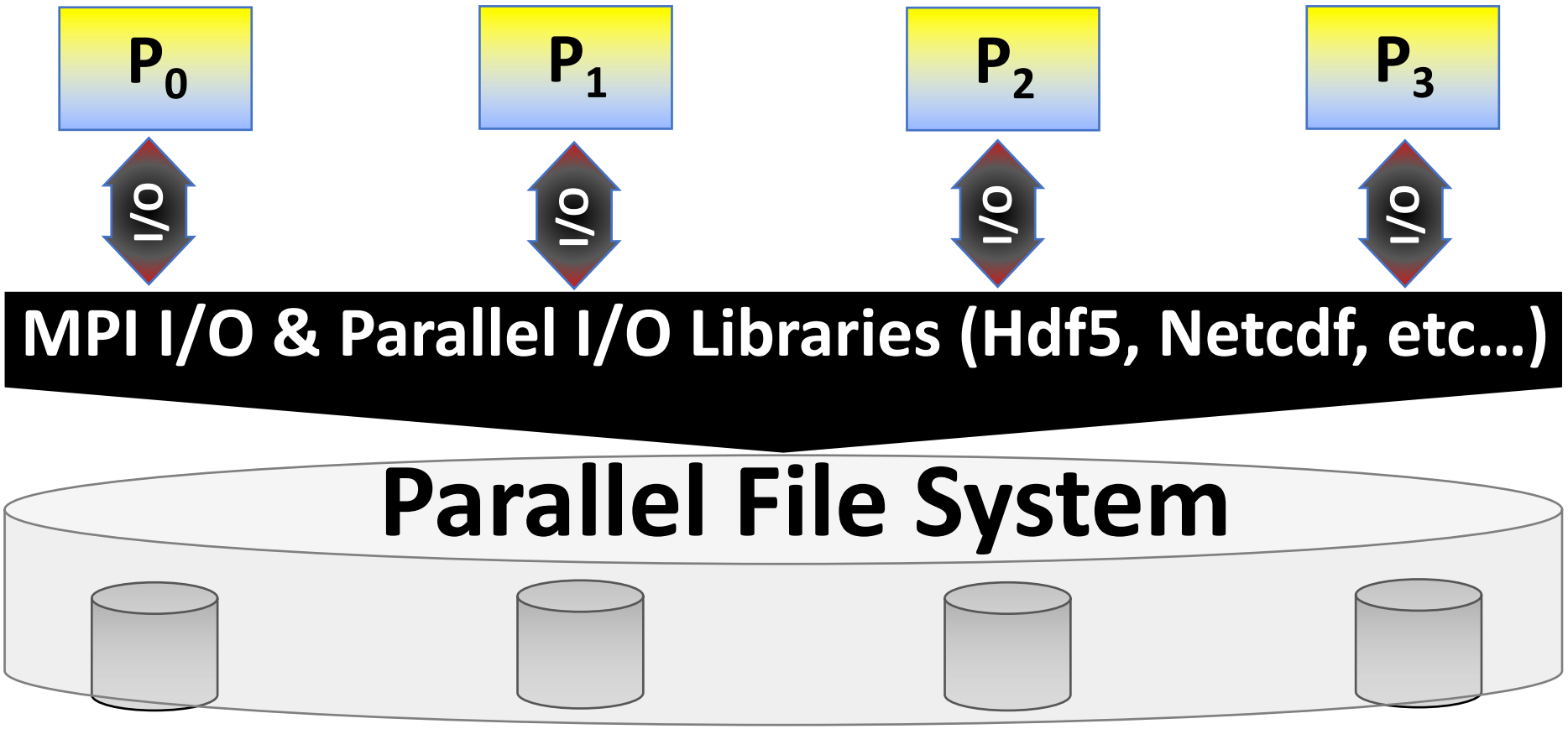
Parallel Data


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: orders 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