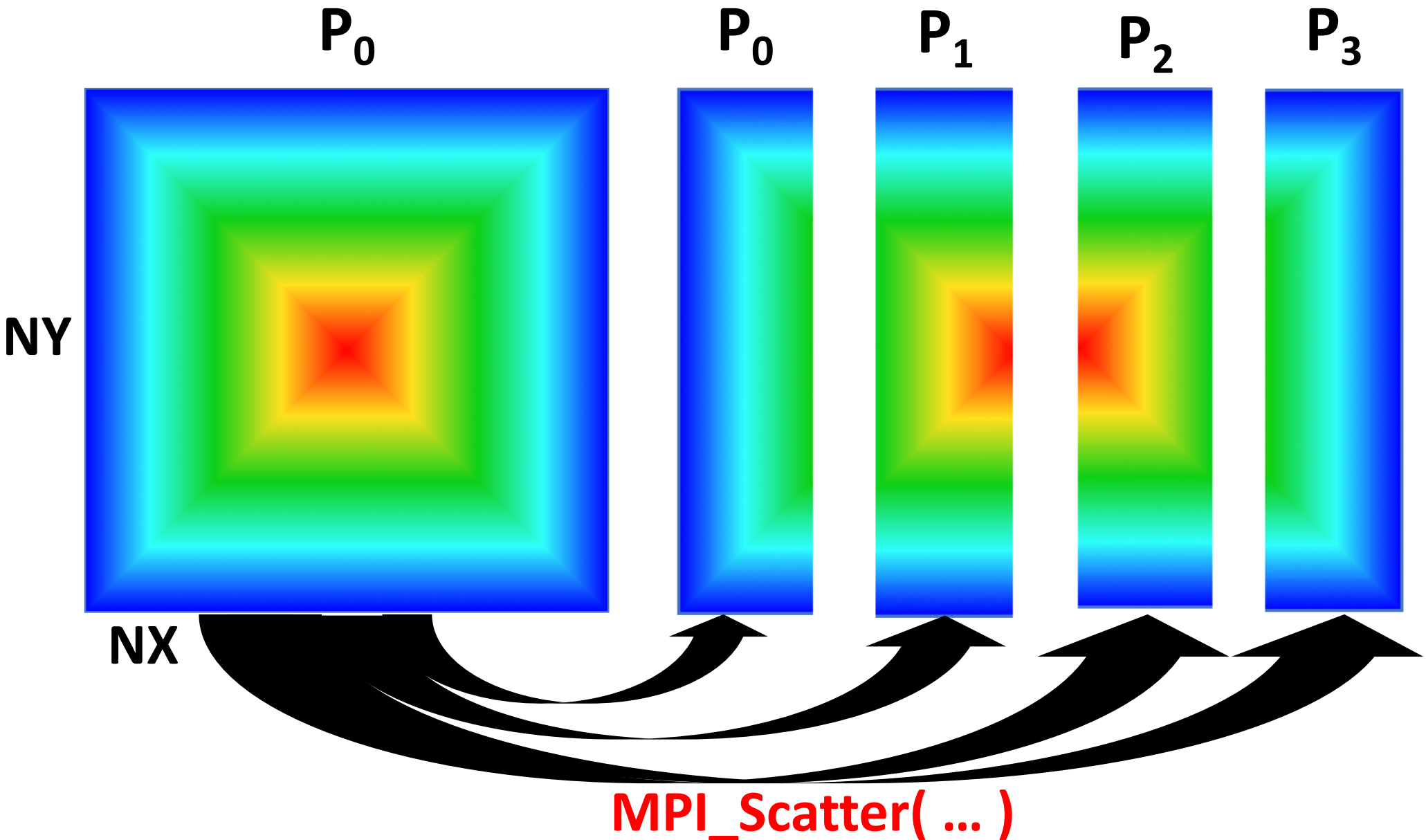
Foundation of Parallel I/O

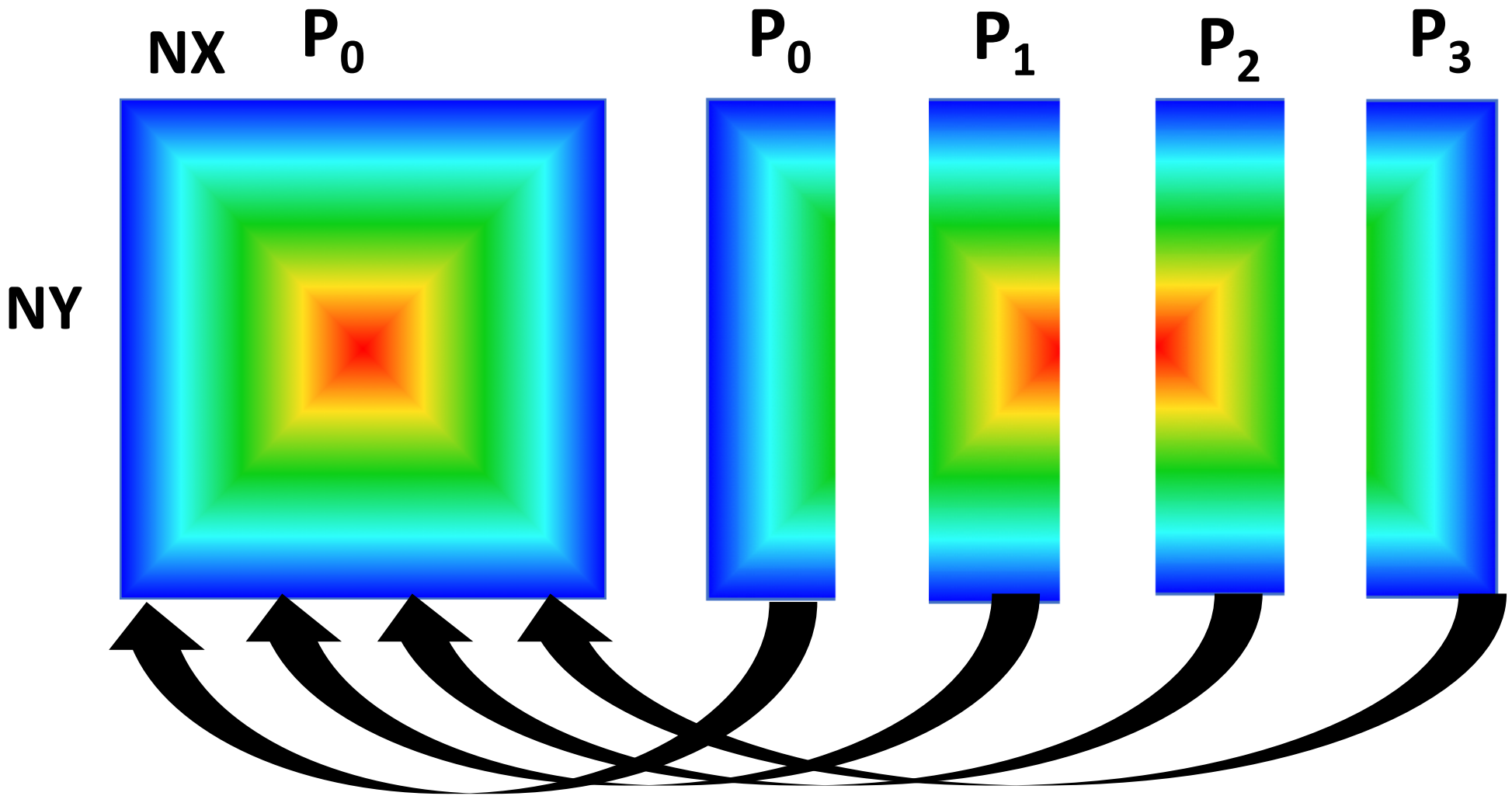
Ivan Girotto – igirotto@ictp.it
International Centre for Theoretical Physics (ICTP)

Data Distribution Recap /1

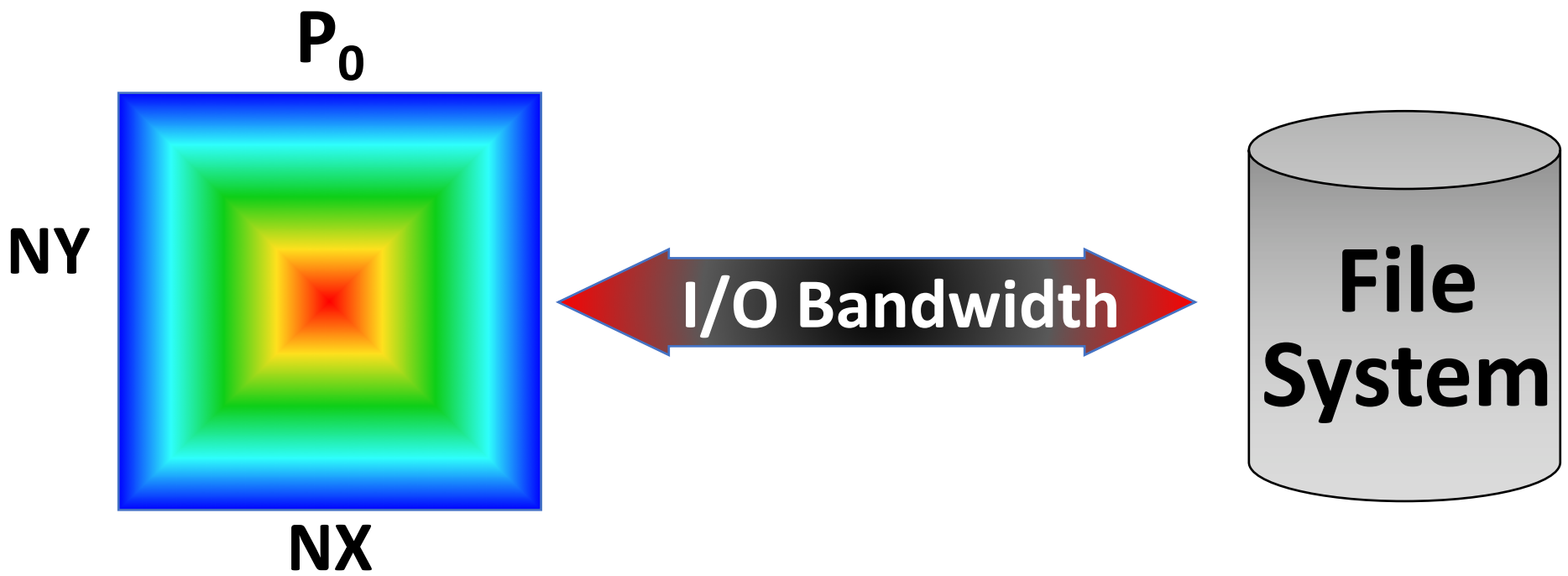
Data Distribution Recap /2



Sequential I/O on Distributed Data /1

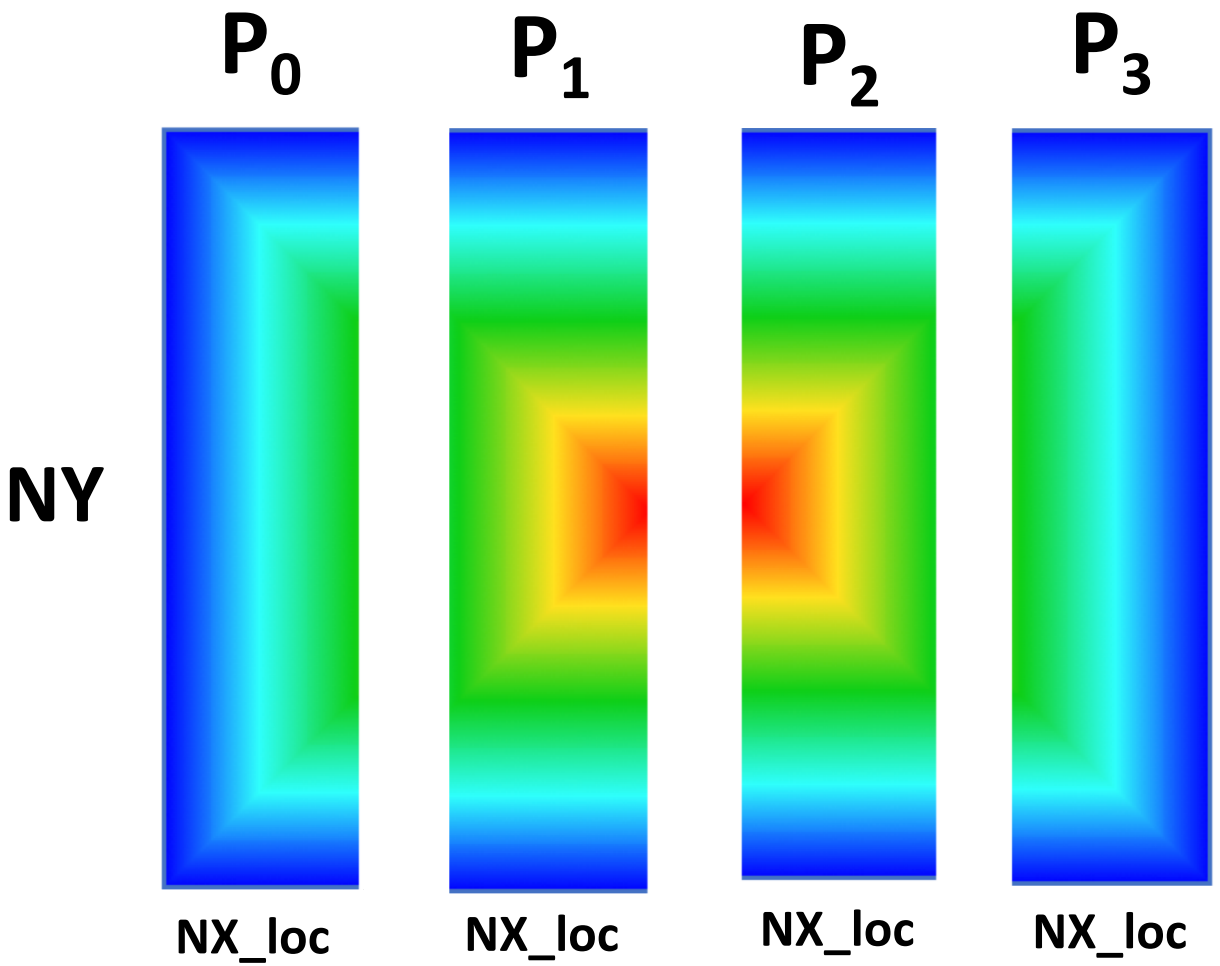


Sequential I/O on Distributed Data /2



Data are finally saved in the disk like on the model.cpp code by using HDF5. By collecting all data on a single process we are implementing a strong data size scaling limitation!

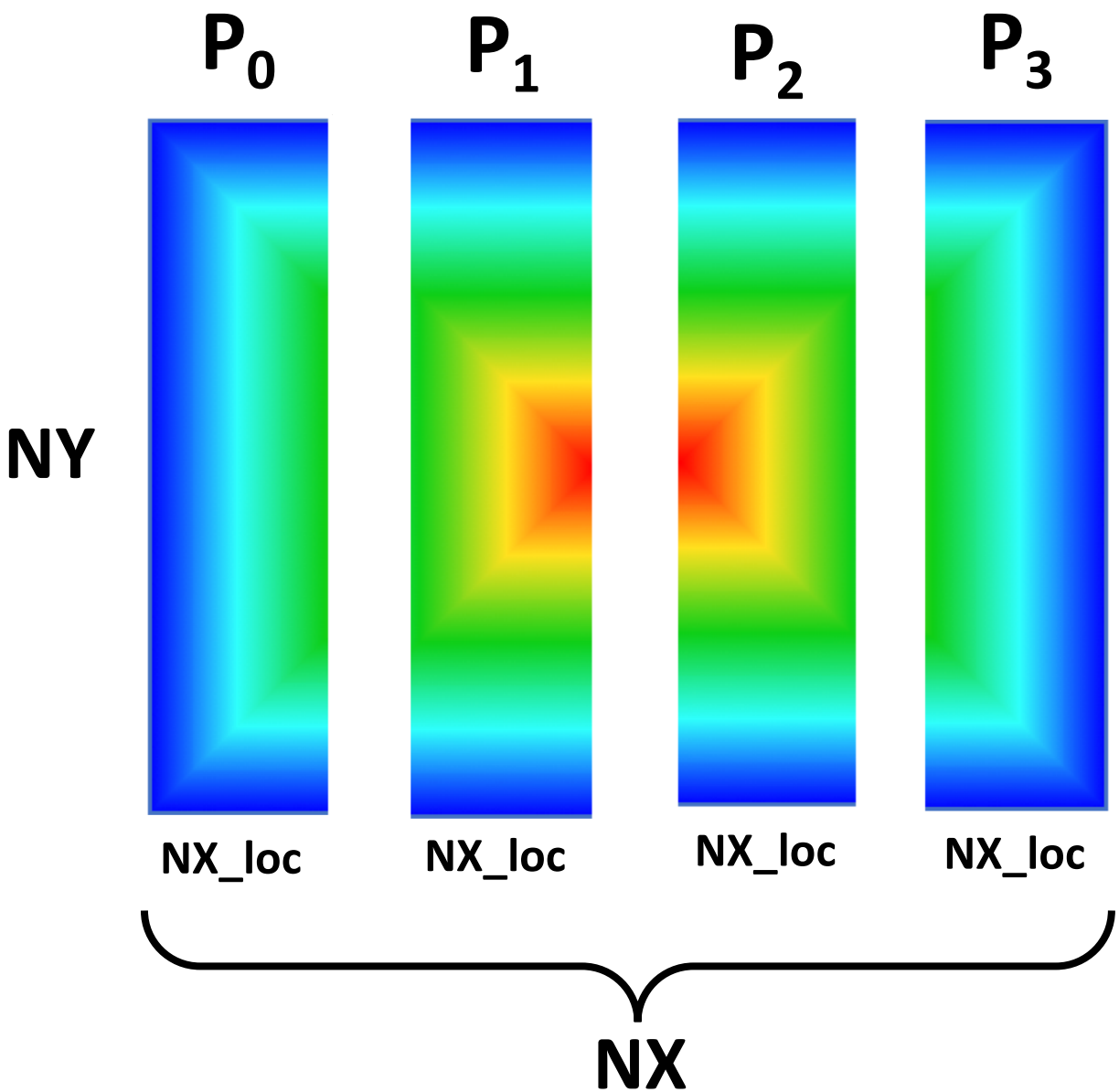
Data Distribution Recap /3



We now want to keep data distributed into memory with never collecting data on a single processor to achieve massively data size scaling!!

Data Distribution Recap /4

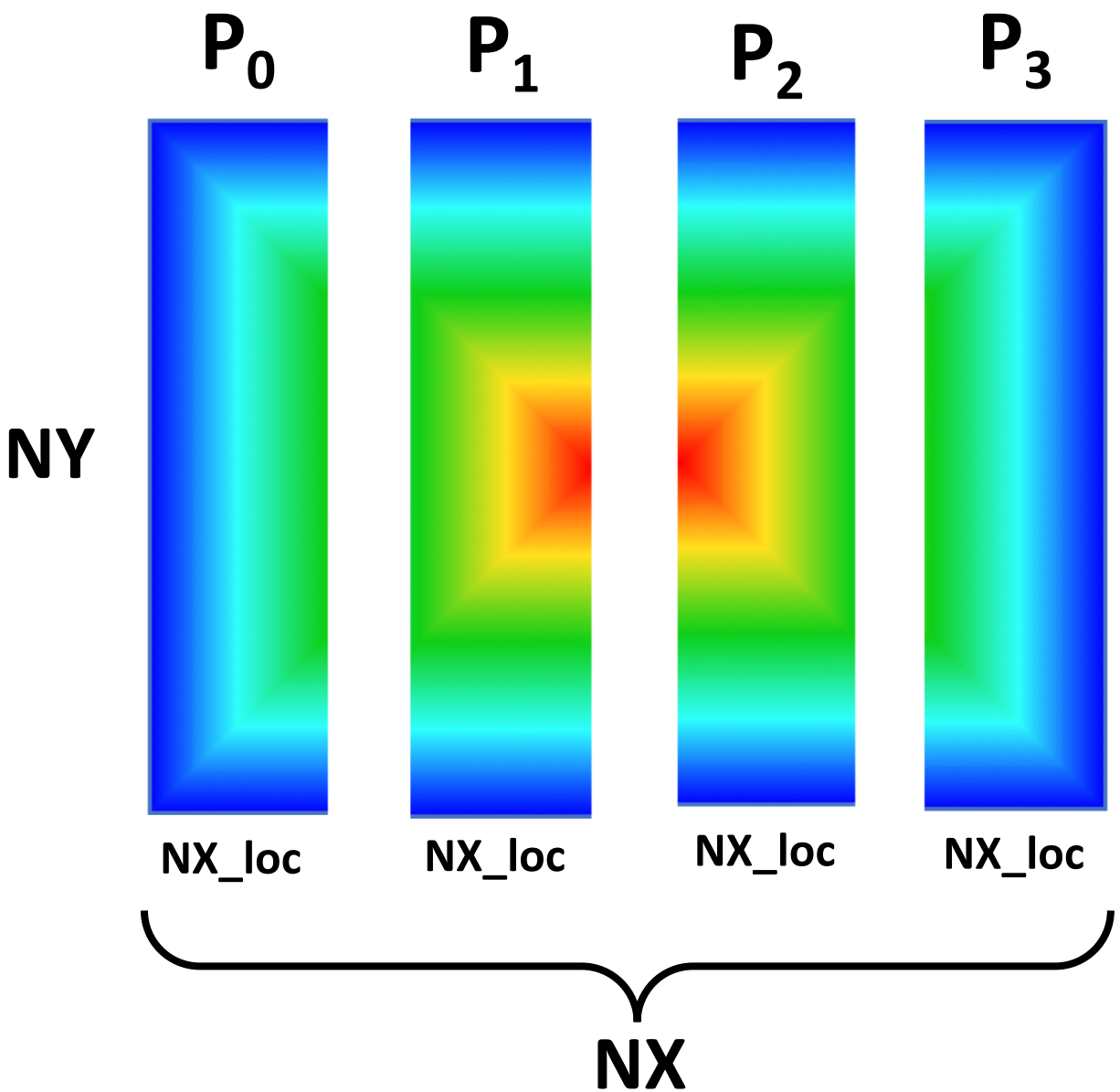
The lucky case: NX % nprocs == 0 !!!




```
NX_loc = NX / nprocs;
nx_start = NX_loc * rank;
```

Data Distribution Recap /6

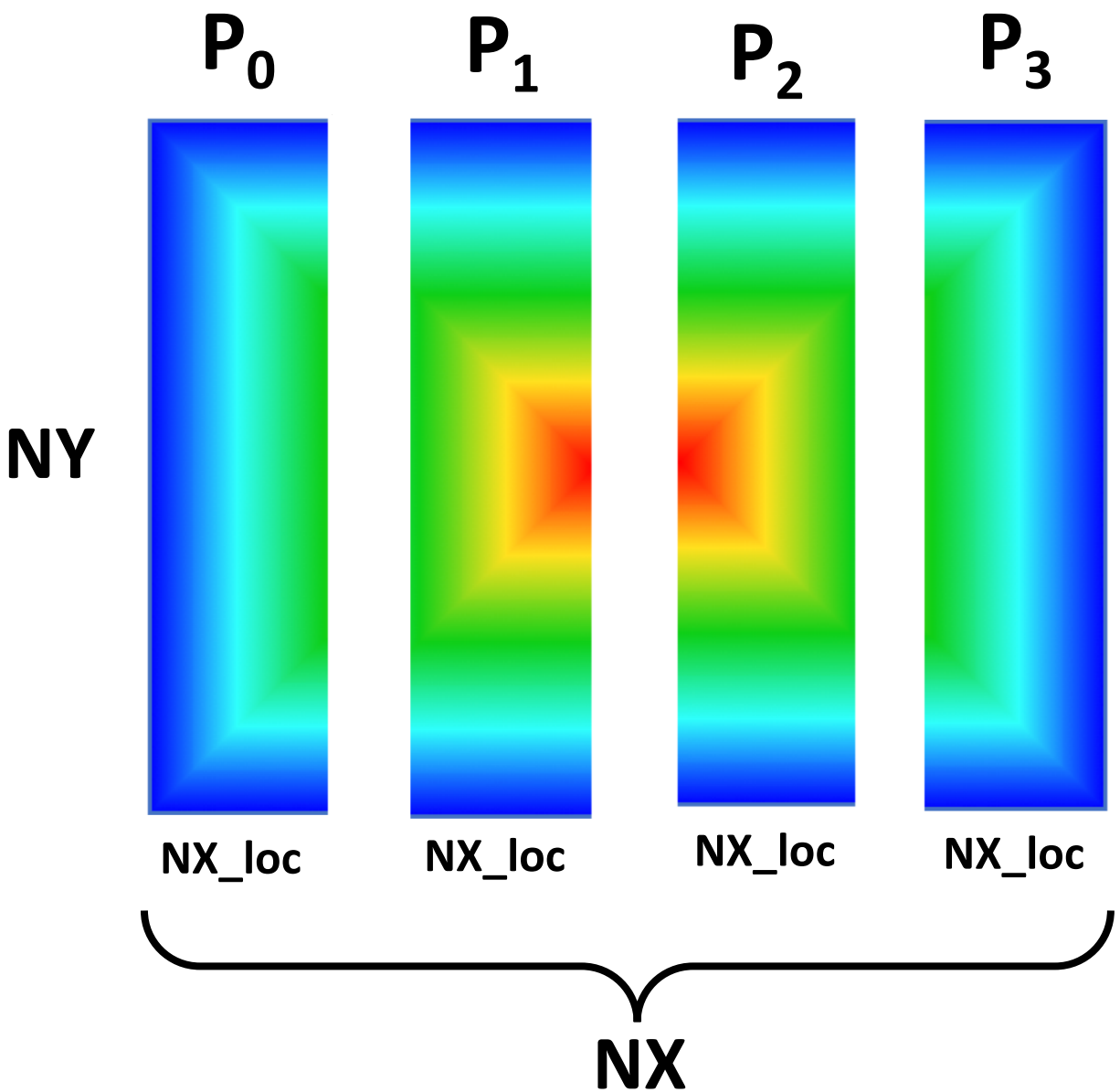
The unlucky case: NX % nprocs != 0 !!!




```
NX_loc = NX / nprocs;
rest = NX % nprocs;
nx_start = NX_loc * rank;
If ( me == nprocs – 1 ) NX_loc += rest;
```

Data Distribution Recap /6

The unlucky case: NX % nprocs != 0 !!!



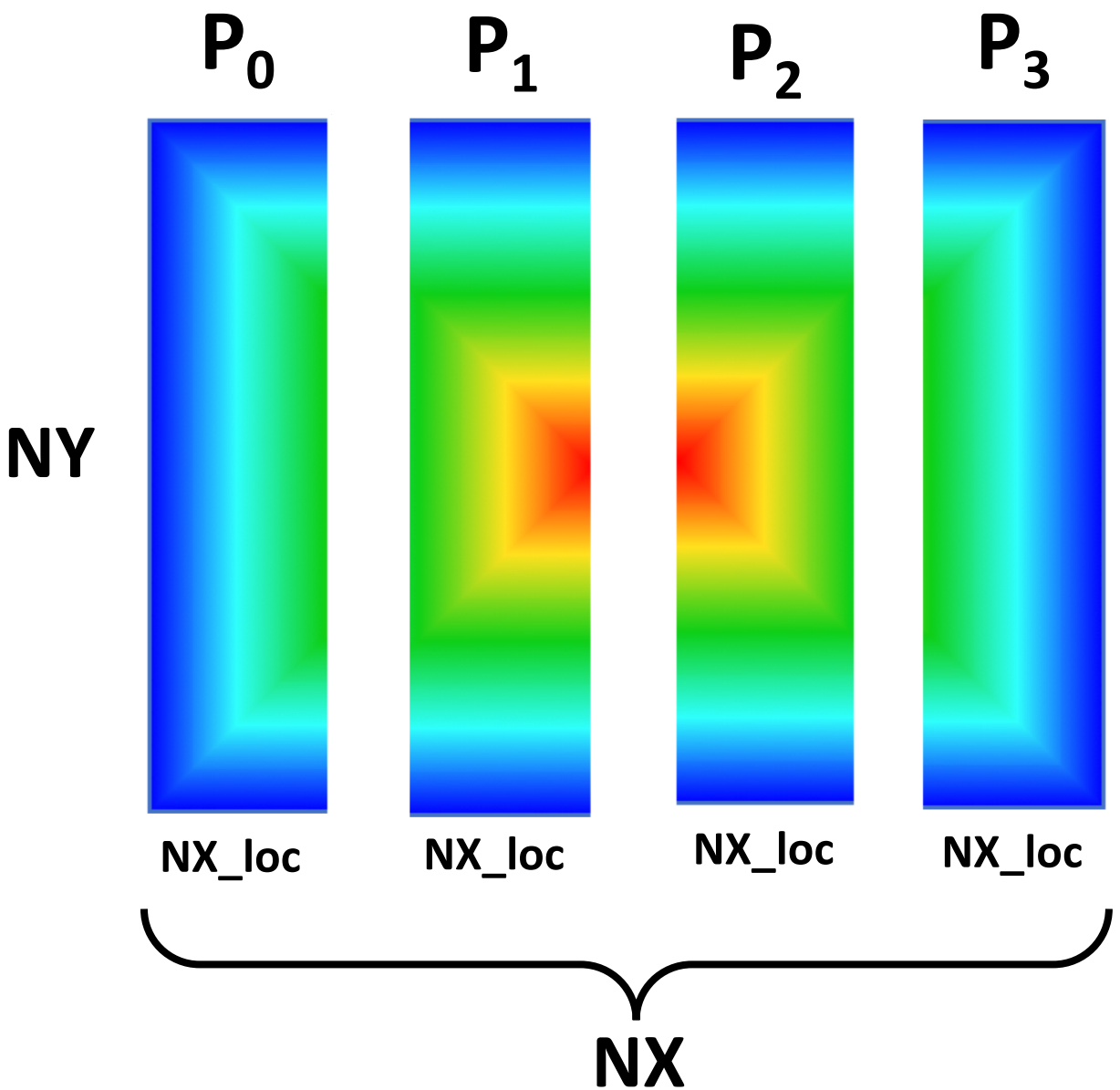

```
NX_loc = NX / nprocs;
rest = NX % nprocs;
nx_start = NX_loc * rank;
If ( me == nprocs – 1 ) NX_loc += rest;
```

Example:

NX = 1400000;
nprocs = 480;

Data Distribution Recap /6

The unlucky case: NX % nprocs != 0 !!!



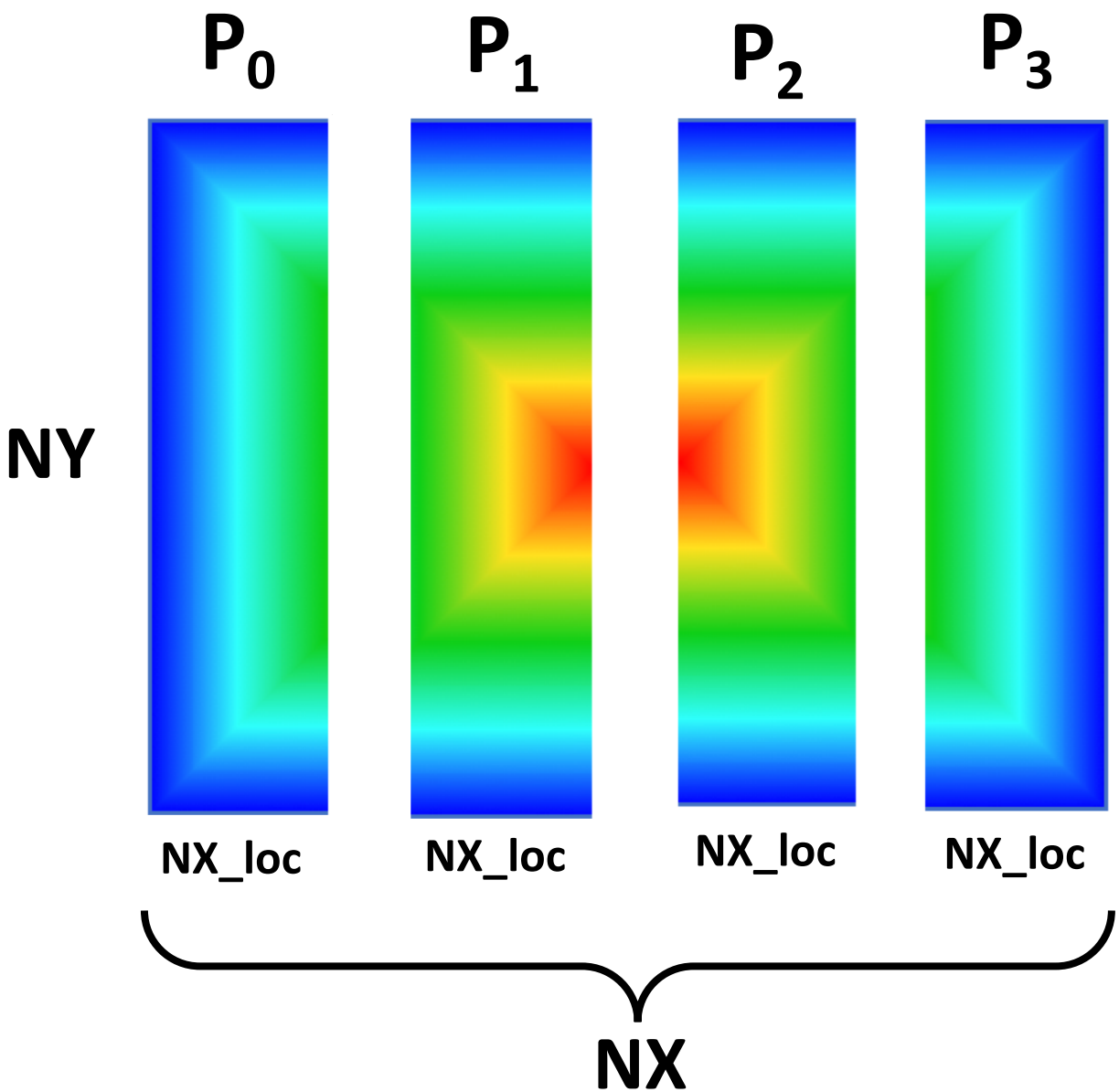

```
NX_loc = NX / nprocs;
rest = NX % nprocs;
nx_start = NX_loc * rank;
If ( me == nprocs – 1 ) NX_loc += rest;
```

Example:

NX = 1400000;
nprocs = 480;
NX_loc = 291

Data Distribution Recap /6

The unlucky case: NX % nprocs != 0 !!!




```
NX_loc = NX / nprocs;
rest = NX % nprocs;
nx_start = NX_loc * rank;
If ( me == nprocs – 1 ) NX_loc += rest;
```

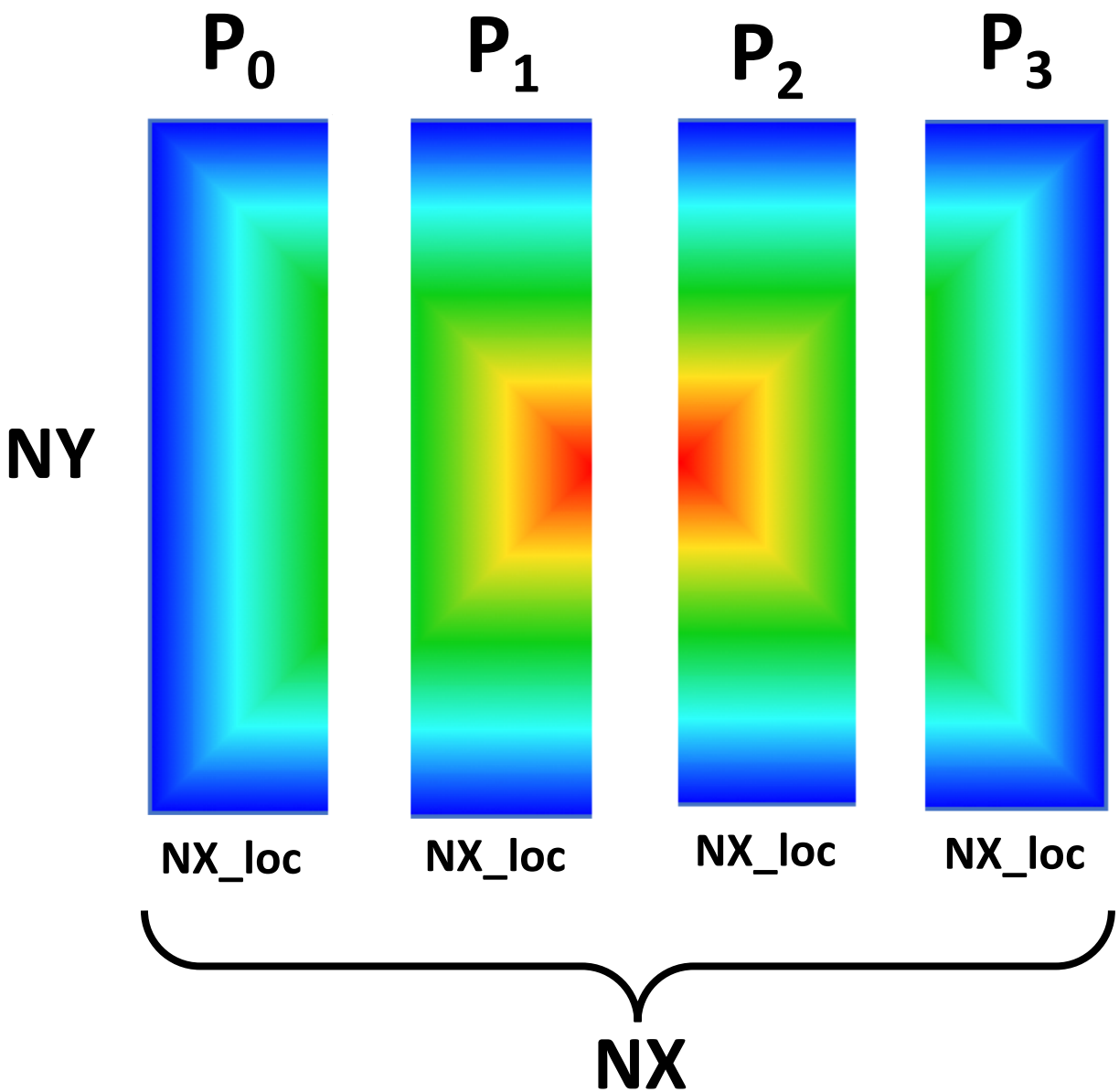
Example:

NX = 1400000;
nprocs = 480;
NX_loc = 291;
rest = 320;
NX_loc (nprocs – 1) = 611!!

NO WAY!!!!

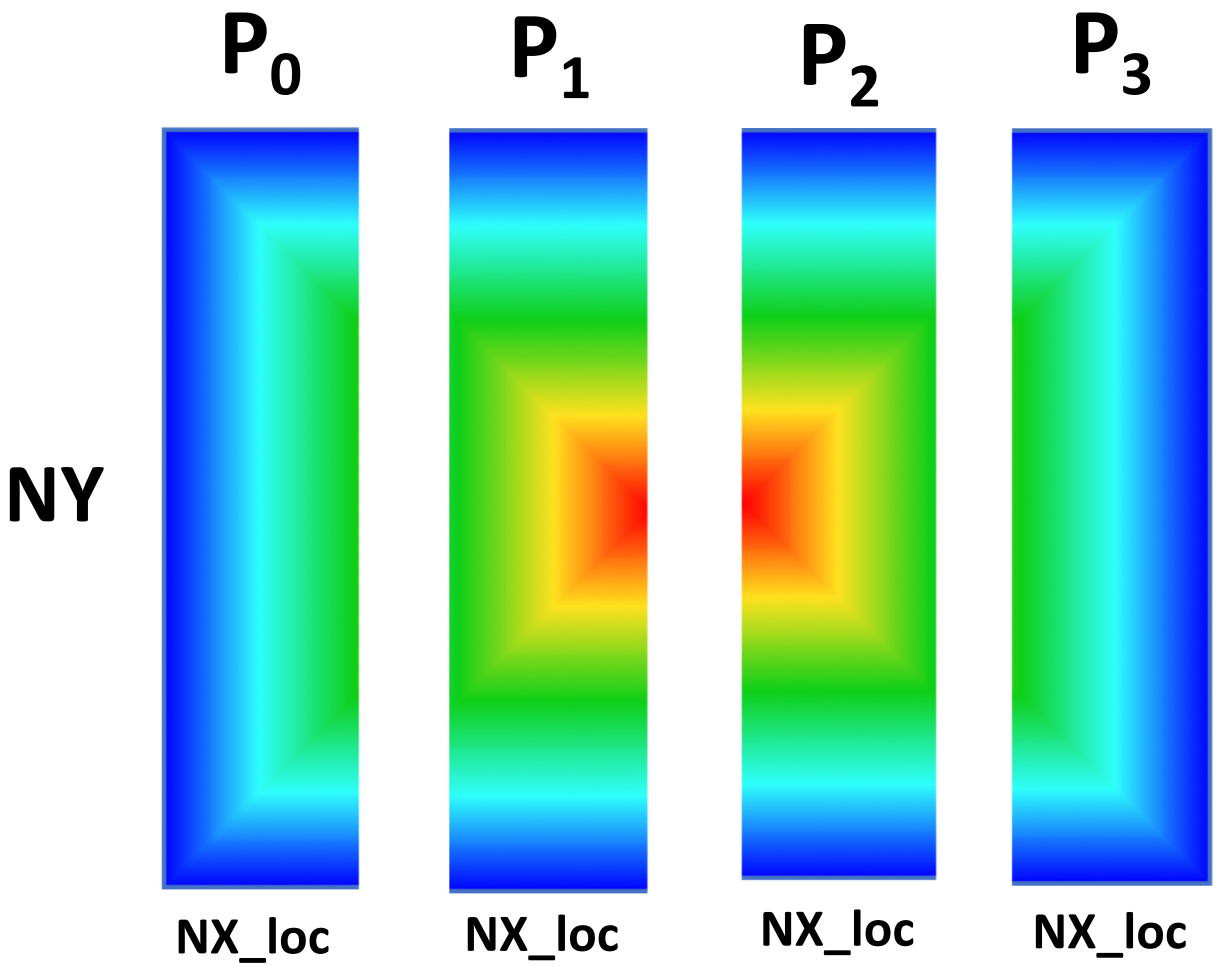
Data Distribution Recap /5

The unlucky case: NX % nprocs != 0 !!!



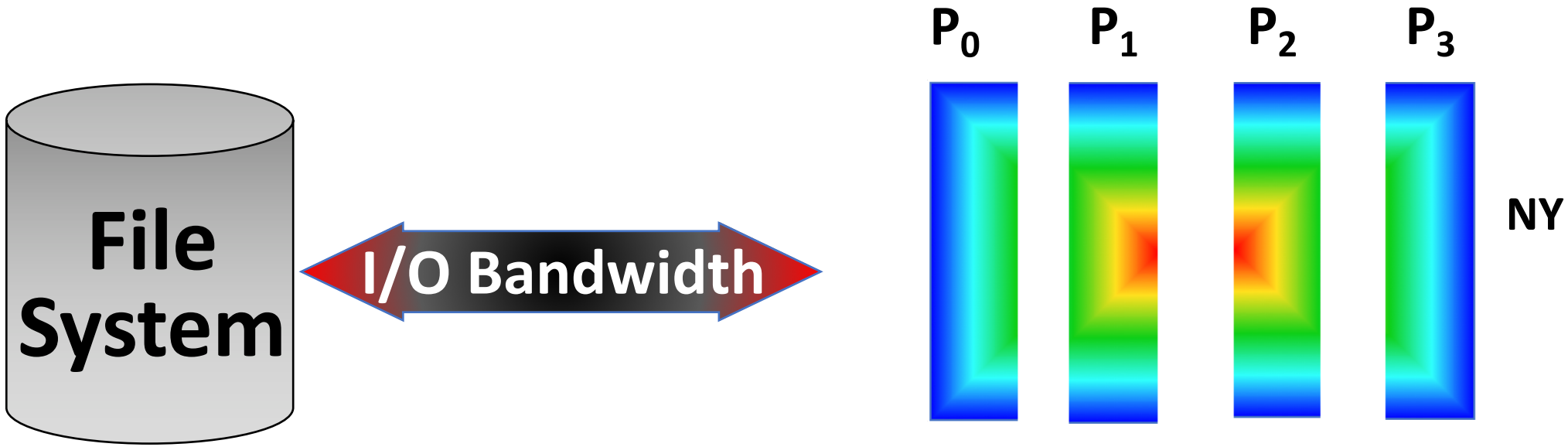

```
NX_loc = NX / nprocs;
rest = NX % nprocs;
If ( me < rest ) NX_loc += 1;
nx_start = NX_loc * rank;
If ( me >= rest ) nx_start += rest;
```

Sequential I/O on Distributed Data /3

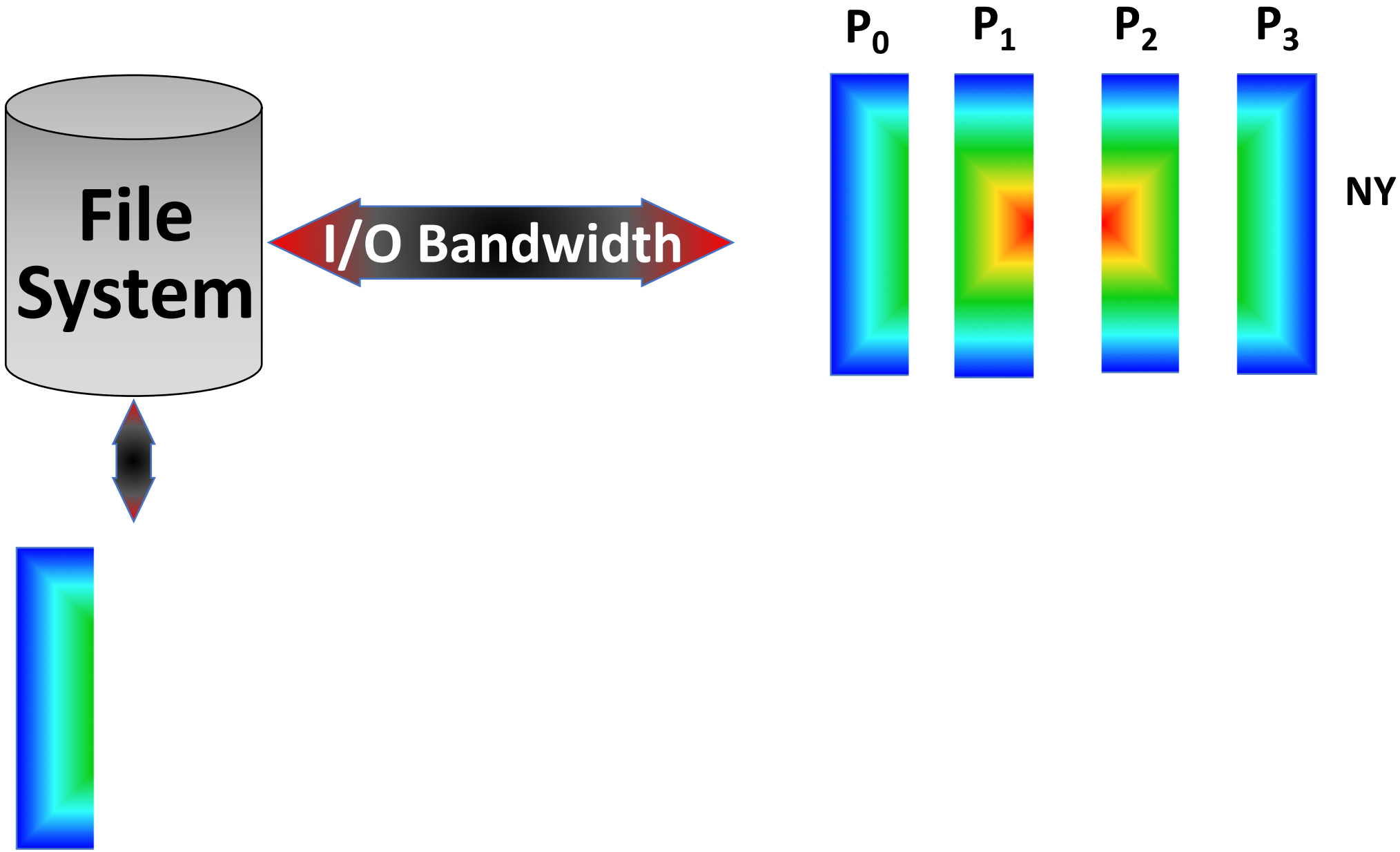


We now want to save data on the disk but we do not have a parallel file system!

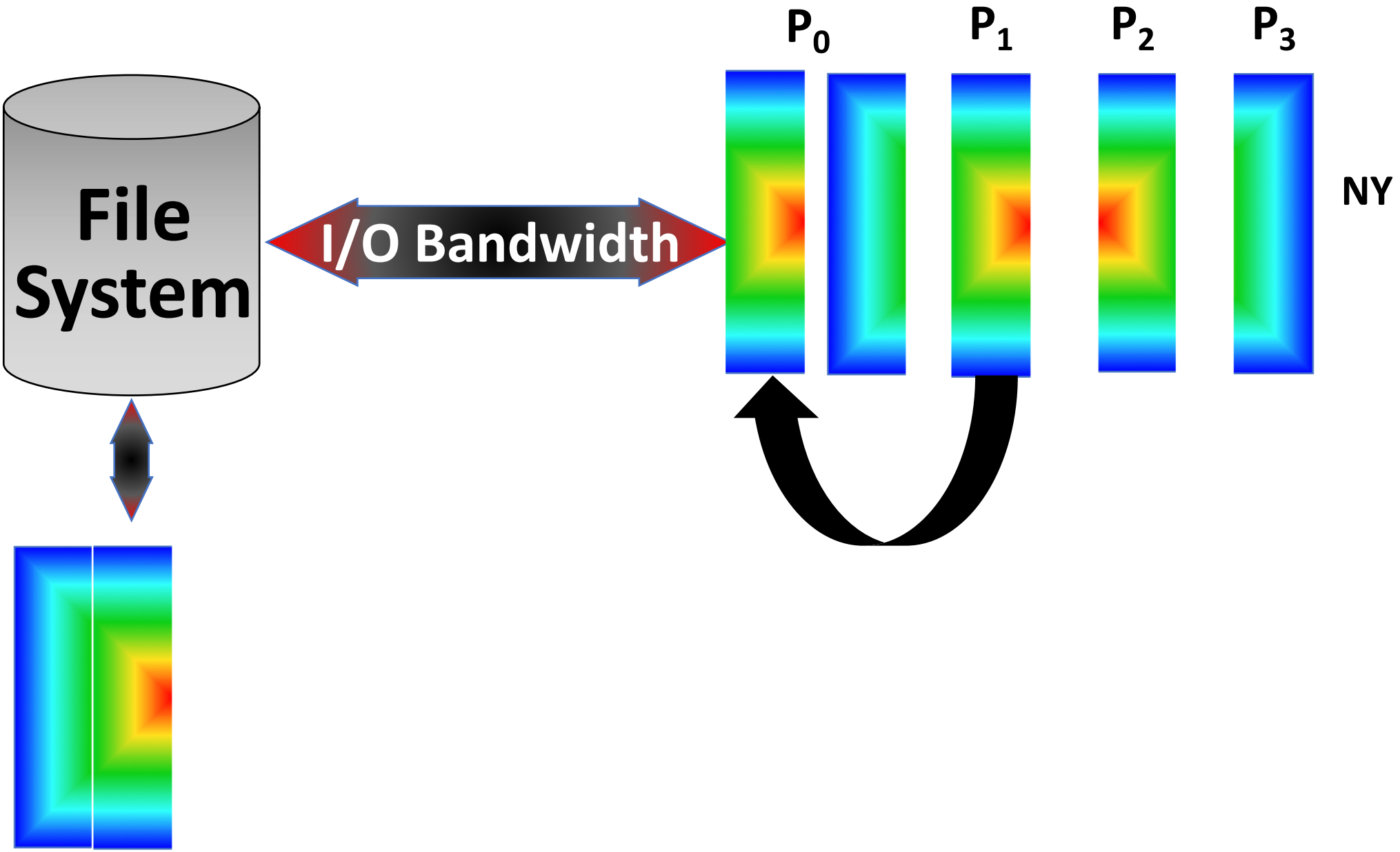
Sequential I/O on Distributed Data /4



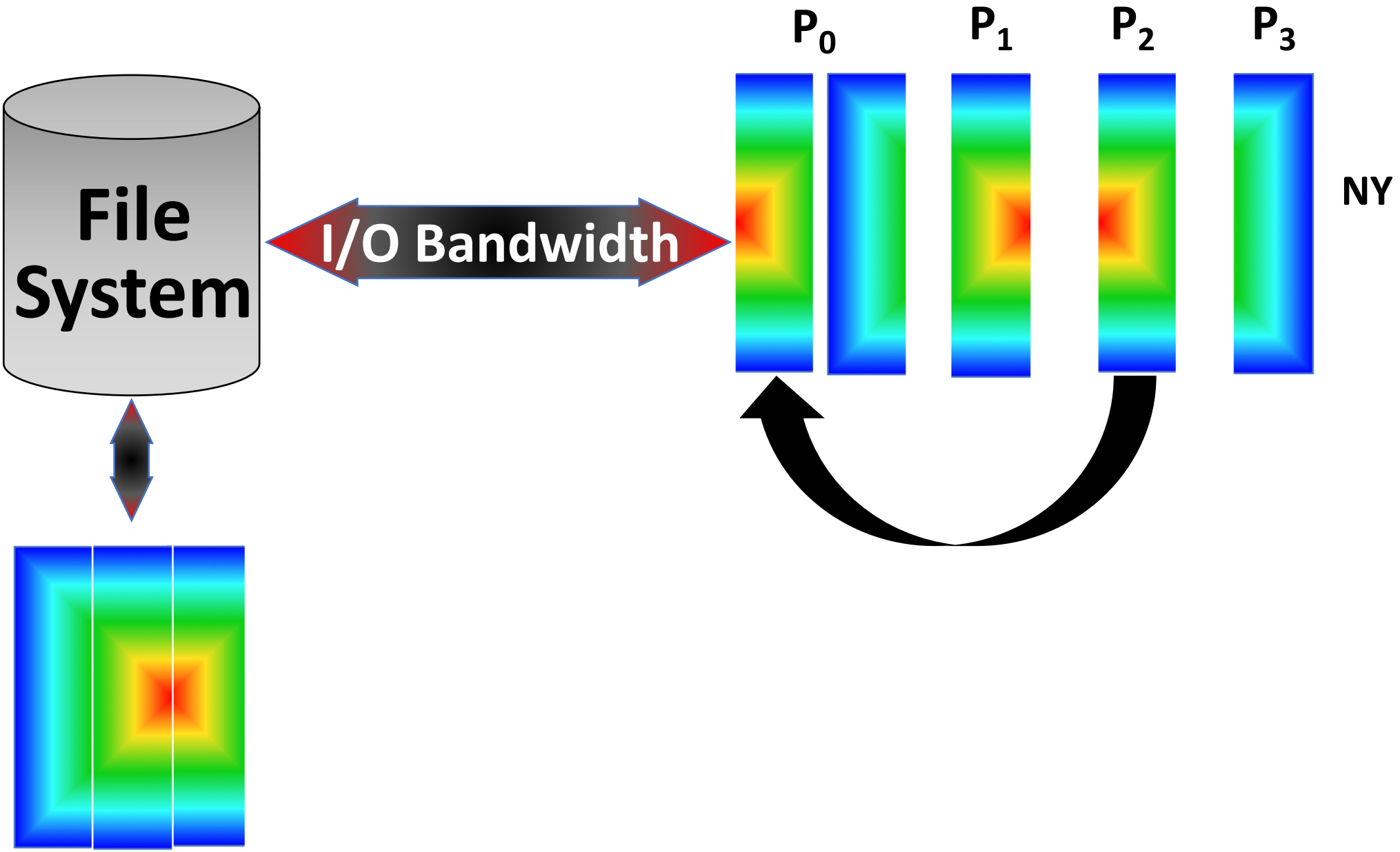
Sequential I/O on Distributed Data /4



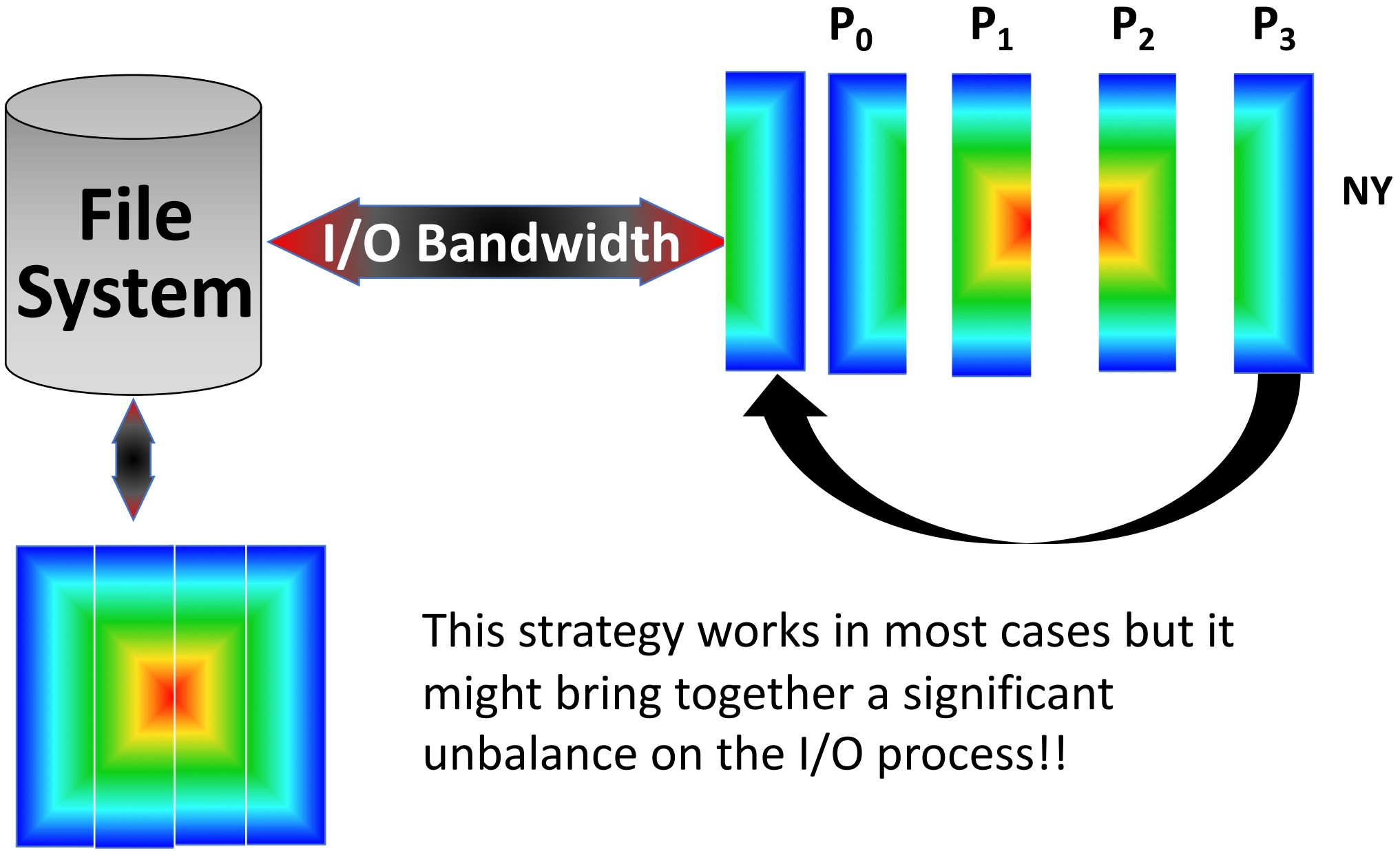
Sequential I/O on Distributed Data /4



Sequential I/O on Distributed Data /4

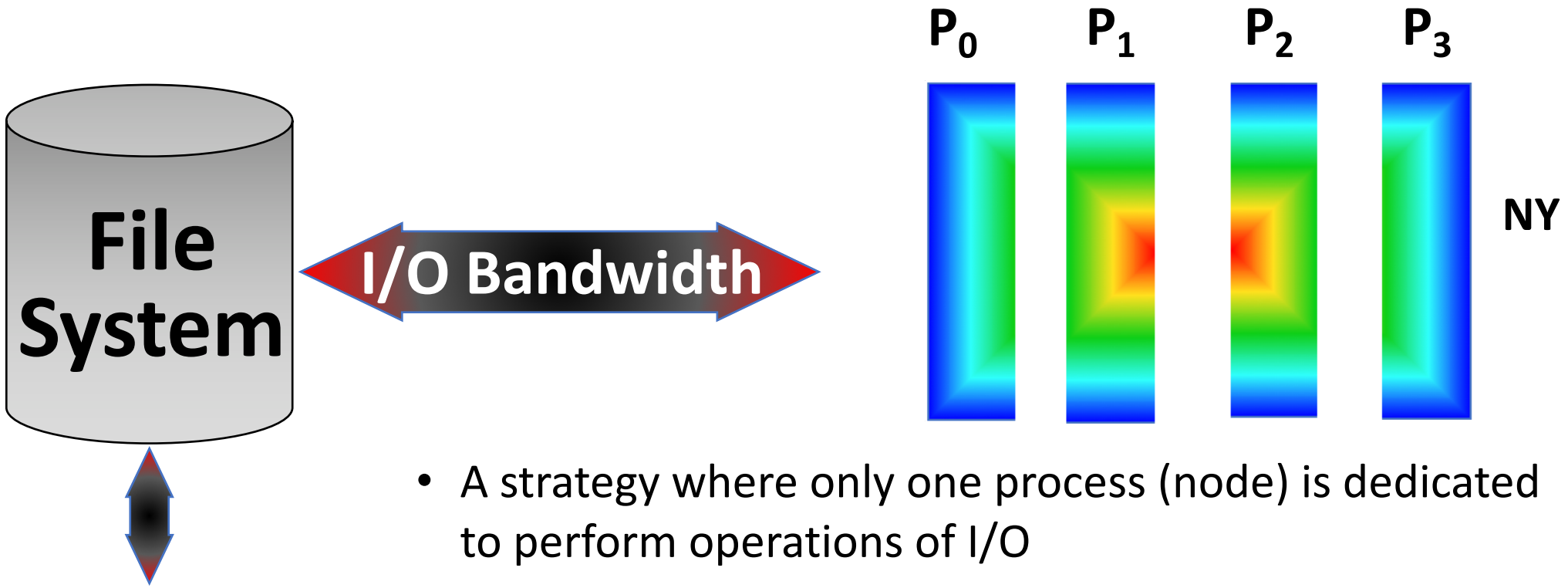


Sequential I/O on Distributed Data /4



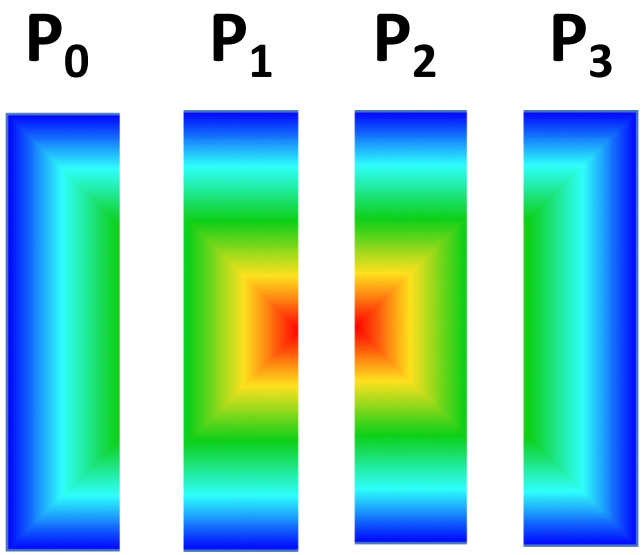
This strategy works in most cases but it might bring together a significant unbalance on the I/O process!!

Sequential I/O on Distributed Data: the I/O node



- A strategy where only one process (node) is dedicated to perform operations of I/O
- On a MPI based parallelization it requires the creation of additional communicators
- The I/O process should be isolated on a single node if allocating much larger memory than others
- For higher scalability multiple I/O can be implemented
  - Non trivial MPI execution model (but possible)

Parallel I/O /1



Parallel I/O: Parallel Reading

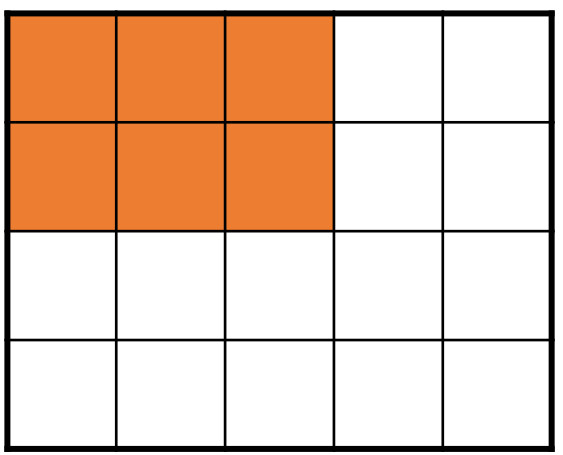
- Parallel Reading is always possible
- No concurrent access on data
- All read all => memory scaling issue!!
- It is really efficient when all read a chunk of data creating a distributed data schema

Parallel I/O: Parallel Reading

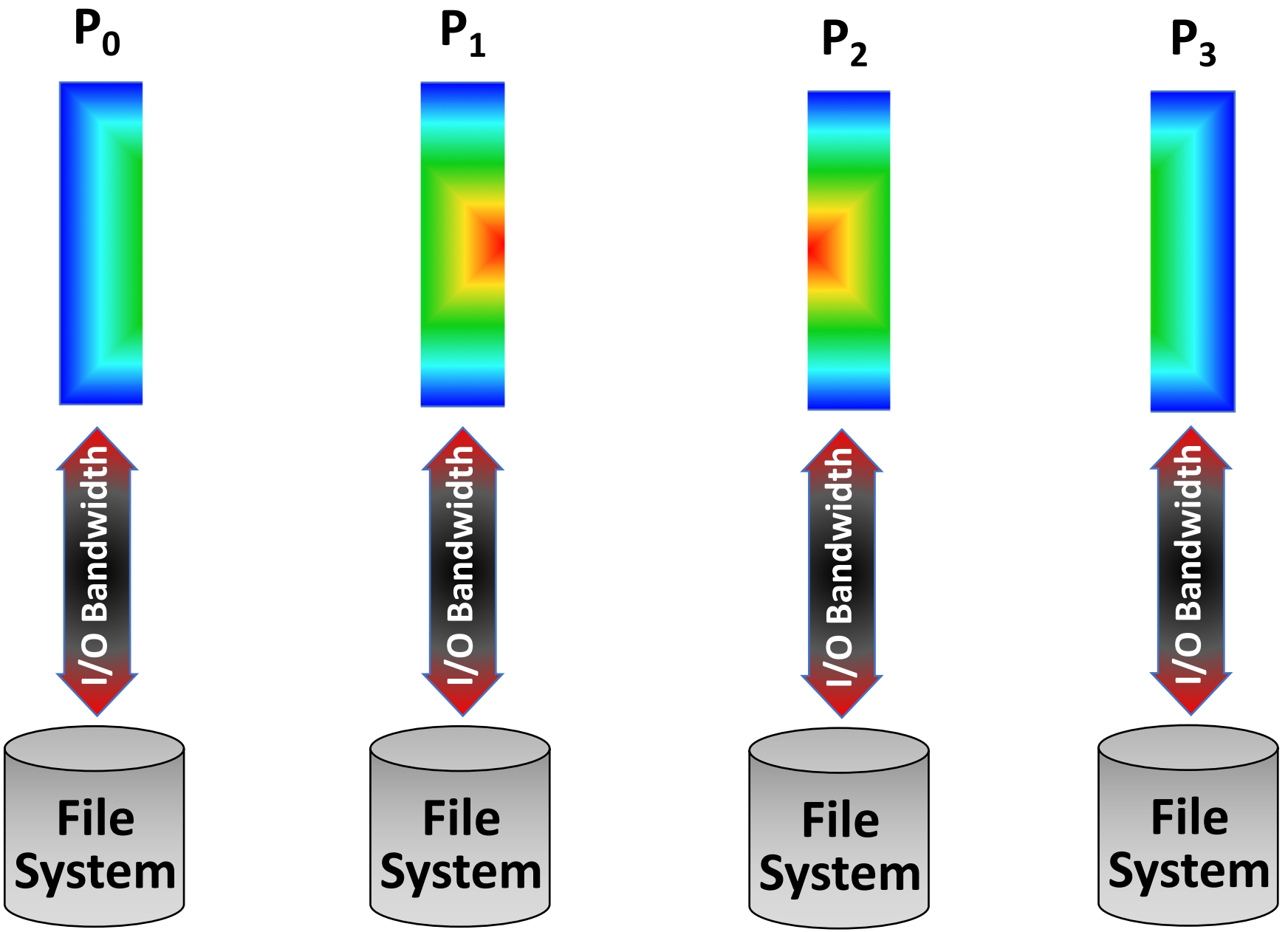
- Parallel Reading is always possible
- No concurrent access on data
- All read all => memory scaling issue!!
- It is really efficient when all read a chunk of data creating a distributed data schema

MPI Data Types!!

MPI_Type_vector(2, 3, 5, MPI_INT, MPI_SUBMAT)



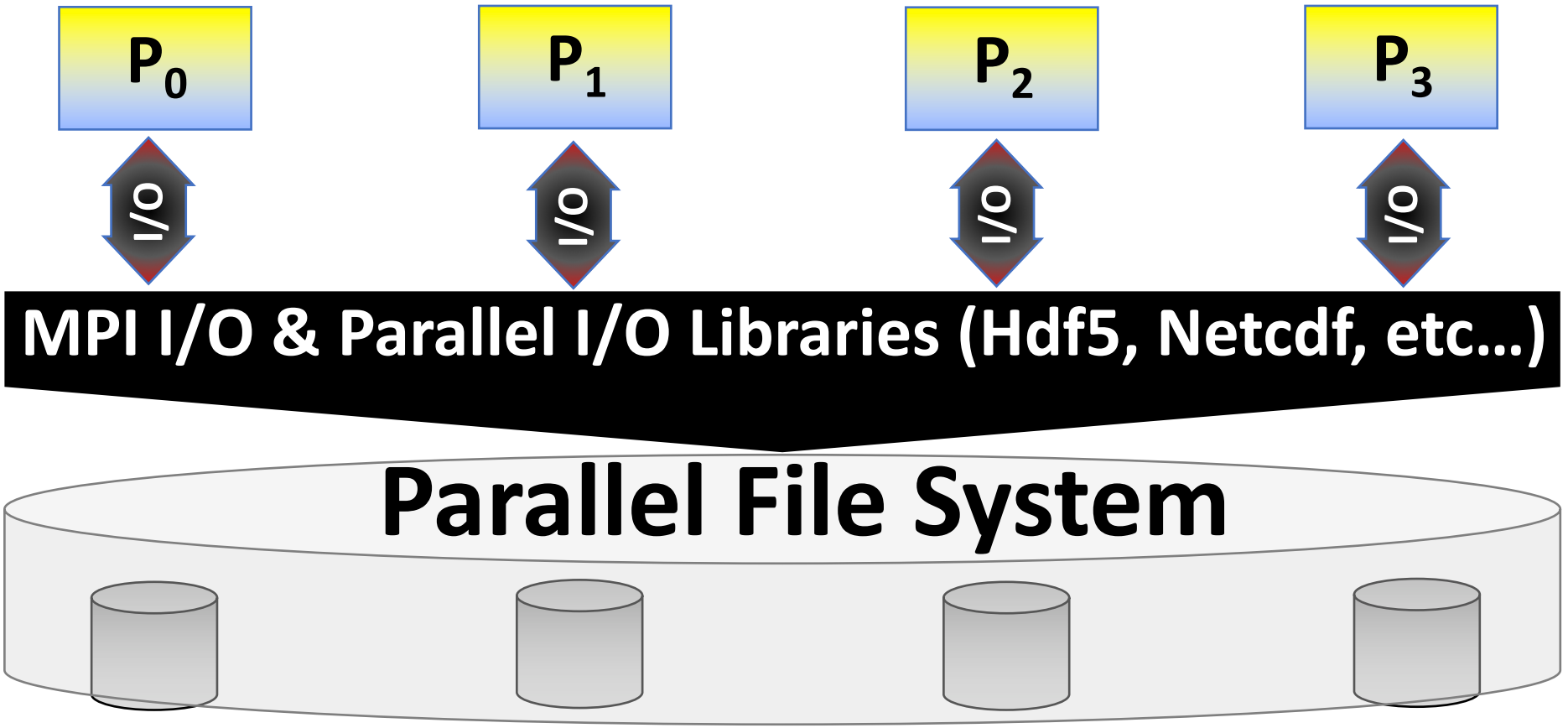
Parallel I/O /1



Parallel I/O /1

- Every process writes is own file
- Easy to implement as long as every file is identified with a unique file name
- Makes really difficult to restart with a different number of processes
- Might create enormous problem at post processing
  - If we need to recreate the whole data-set we need to read the data again and do what we were supposed to do initially when data were first moving from memory to disk
  - Possible creation of an enormous number of files. You could create a problem not only to your self but to the whole infrastructure

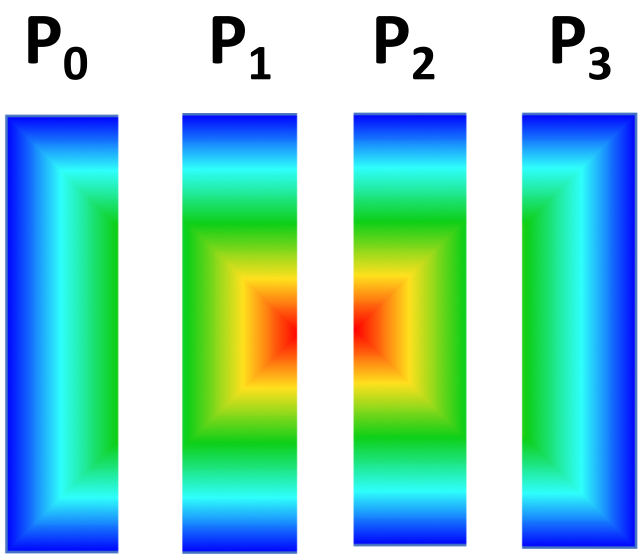
Parallel I/O /2



- The Parallel File system provide the utility to write a file from distributed processes on the storage system
- The intermediate level is essential to avoid learning how to handle concurrent access to shared file pointes :-o

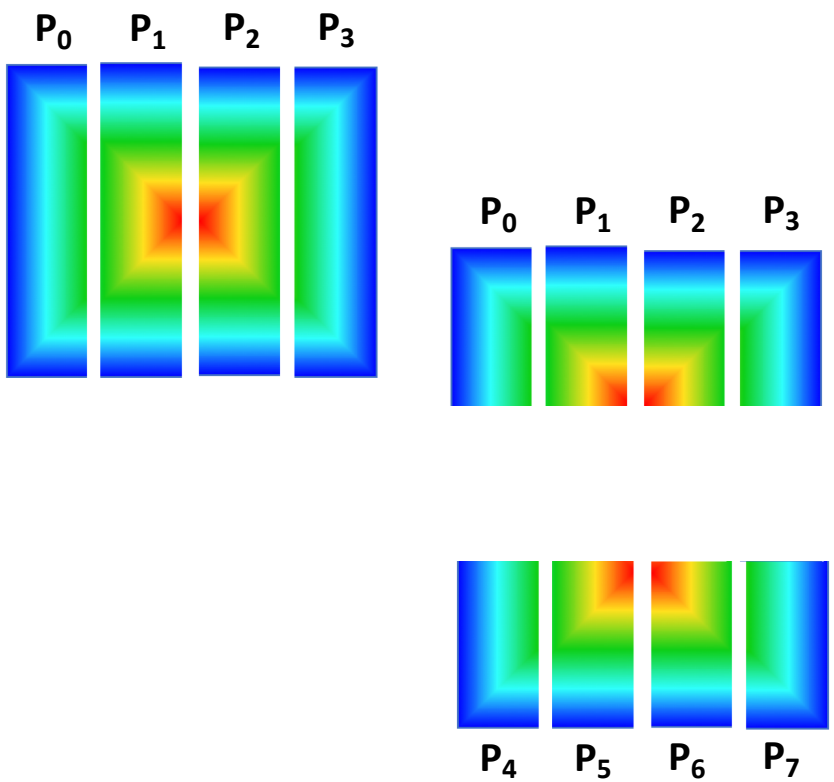
Parallel I/O: MPI I/O

- Use of MPI routines for parallel I/O
- Interfaces for handling shared access on the same file
- Developers must implement a way to store data accordingly to a given data layout => MPI data types helps!



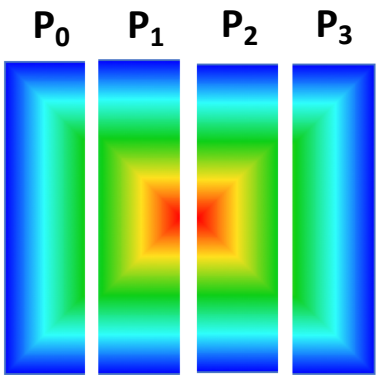
Parallel I/O: MPI I/O

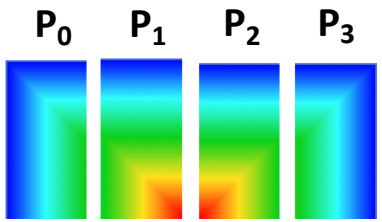
- Use of MPI routines for parallel I/O
- Interfaces for handling shared access on the same file
- Developers must implement a way to store data accordingly to a given data layout => MPI data types helps!

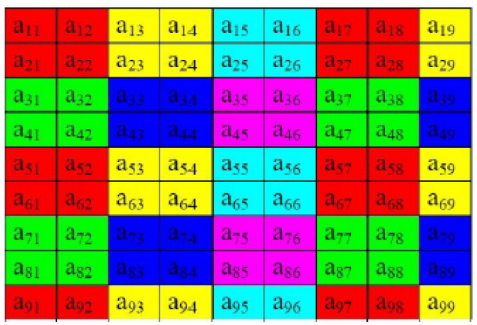


Parallel I/O: MPI I/O

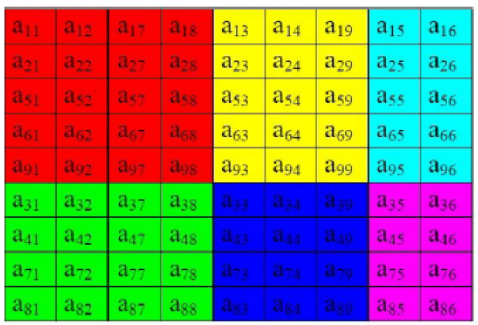
- Use of MPI routines for parallel I/O
- Interfaces for handling shared access on the same file
- Developers must implement a way to store data accordingly to a given data layout => MPI data types helps!



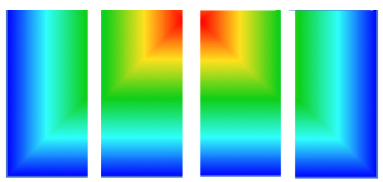




Logical View (Matrix)

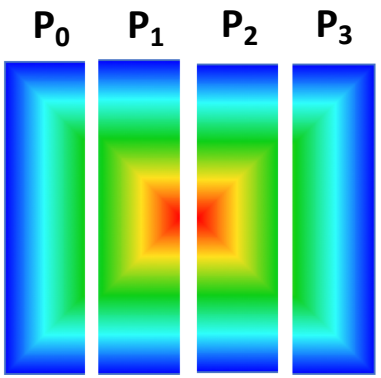


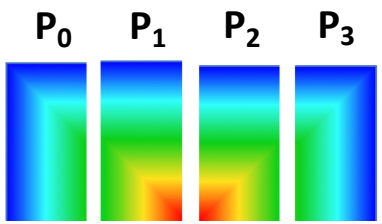
Local View (CPUs)

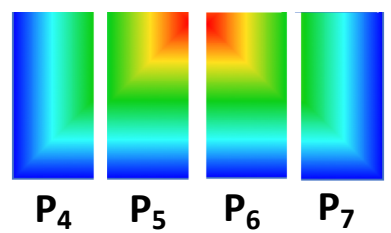


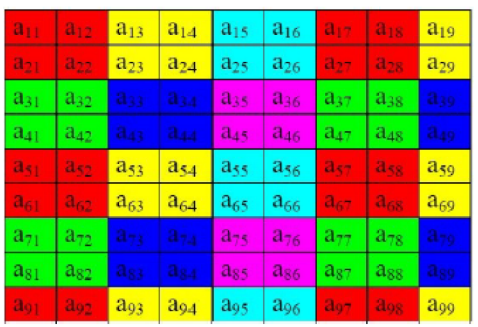
Parallel I/O: MPI I/O

- Use of MPI routines for parallel I/O
- Interfaces for handling shared access on the same file
- Developers must implement a way to store data accordingly to a given data layout => MPI data types helps!

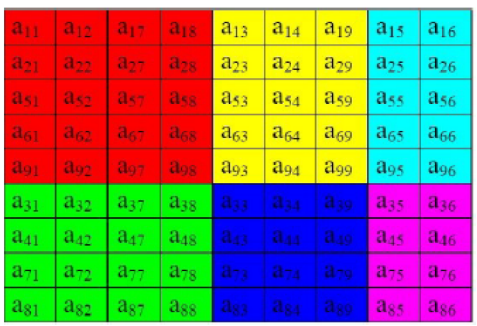








Logical View (Matrix)



Local View (CPUs)

Good Luck with 3D!!!!

Parallel I/O: HDF5

- You don’t need much more of what you have seen yesterday!!
- Add a call to to the H5Pset_dxpl_mpio to tell HDF5 you are going to use Parallel I/O

```
plist_id = H5Pcreate (H5P_FILE_ACCESS);

hdf5_status = H5Pset_fapl_mpio (plist_id, MPI_COMM_WORLD, MPI_INFO_NULL);
```

- The hyperslab is still used to described the distributed dataset
- Remember to define a good balance about what to write in every single file (i.e., trajectories, temporal dynamics, time evolution, ecc.)
  - Few files containing large data
  - A lot of files containing few data

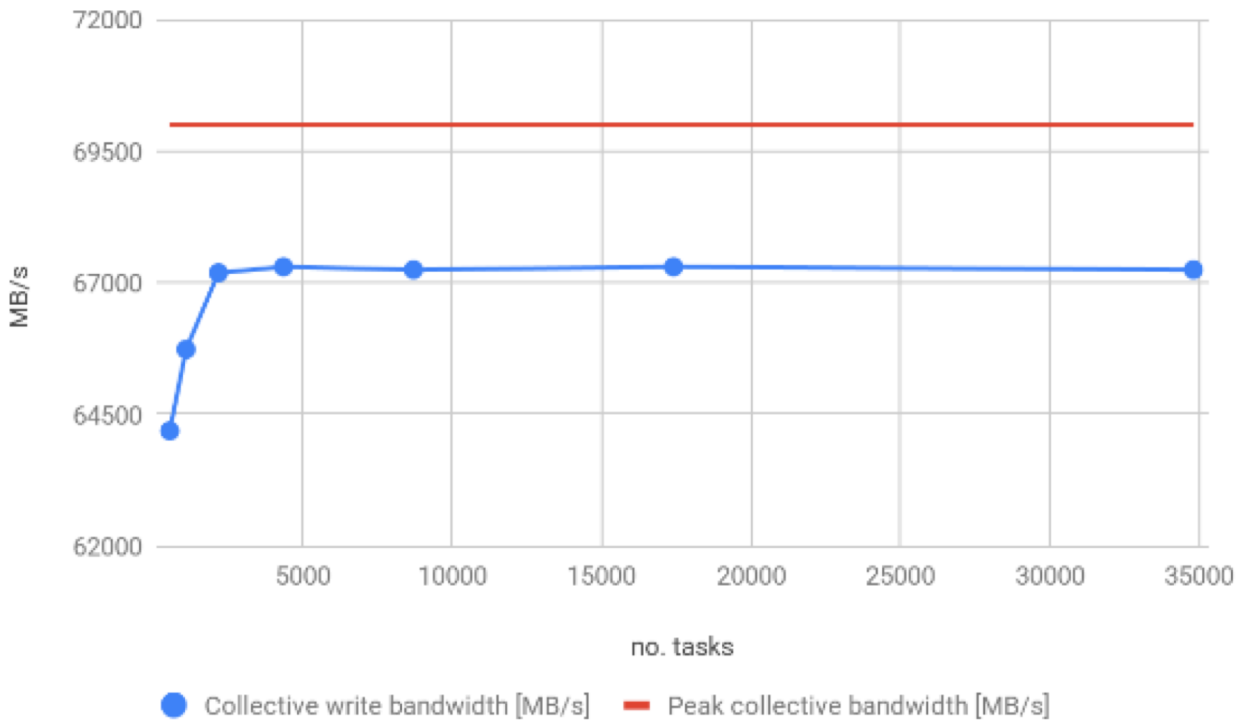
Parallel I/O: possible approaches

1. Open and close the file every time for writing/reading data (inefficient)
2. Open and close the a different file every time for writing data (inefficient but sometime a good compromise)
3. Open the file for several writing/reading access data
   - ❖ In general more efficient because aiding the I/O system buffering
   - ❖ In writing file can become really big (hard to post-process)
4. Buffer in memory several I/O writing to perform less accesses to the file system with larger dumps from memory to disk (really efficient but requires experience for maintaining a scalable design)

Parallel I/O: performances

WHY HDF5

- API beyond POSIX
- So far the bandwidth usage has shown to be an excellent ratio of the theoretical peak
- HDF5 + MPI-IO is proving to be an efficient, portable and convenient abstraction for high-throughput workloads



* Curtesy of Carlo Cavazzoni (CINECA)