The Abdus Salam International Centre for Theoretical Physics

Best practices of parallel programming for HPC

Ivan Girotto – igirotto@ictp.it

International Centre for Theoretical Physics (ICTP)

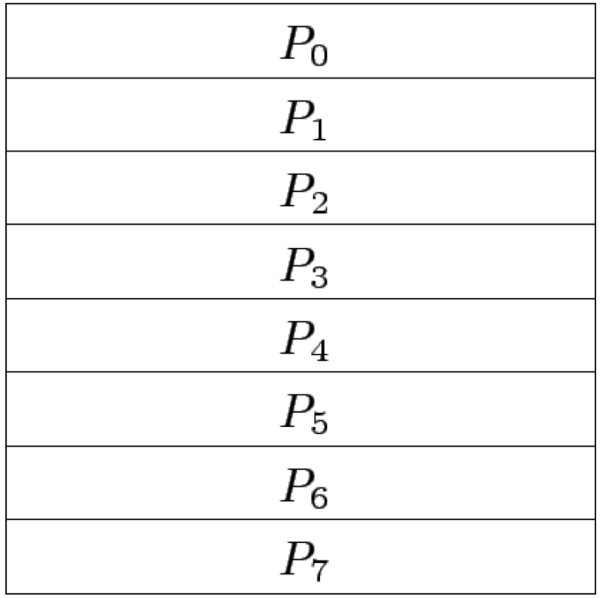
Parallelism - 101

- there are two main reasons to write a parallel program:
  - access to larger amount of memory (aggregated, going bigger)
  - reduce time to solution (going faster)

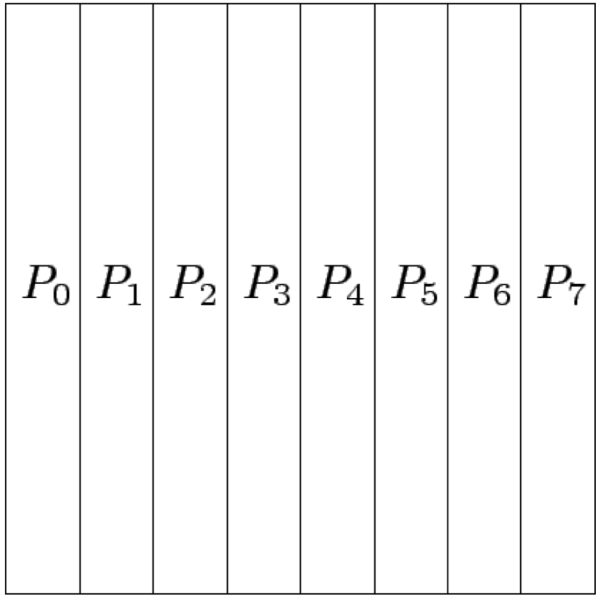
Static Data Partitioning

The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution



column-wise distribution

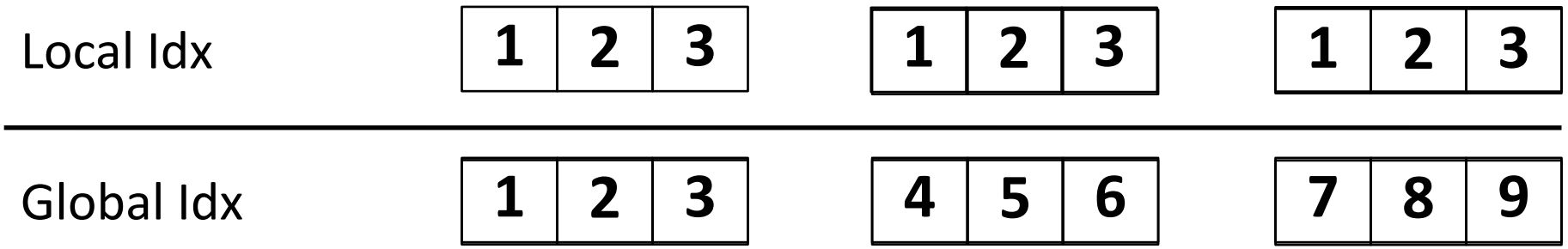


Distributed Data Vs Replicated Data

- Replicated data distribution is useful if it helps to reduce the communication among process at the cost of bounding scalability
- Distributed data is the ideal data distribution but not always applicable for all data-sets
- Usually complex application are a mix of those techniques => distribute large data sets; replicate small data

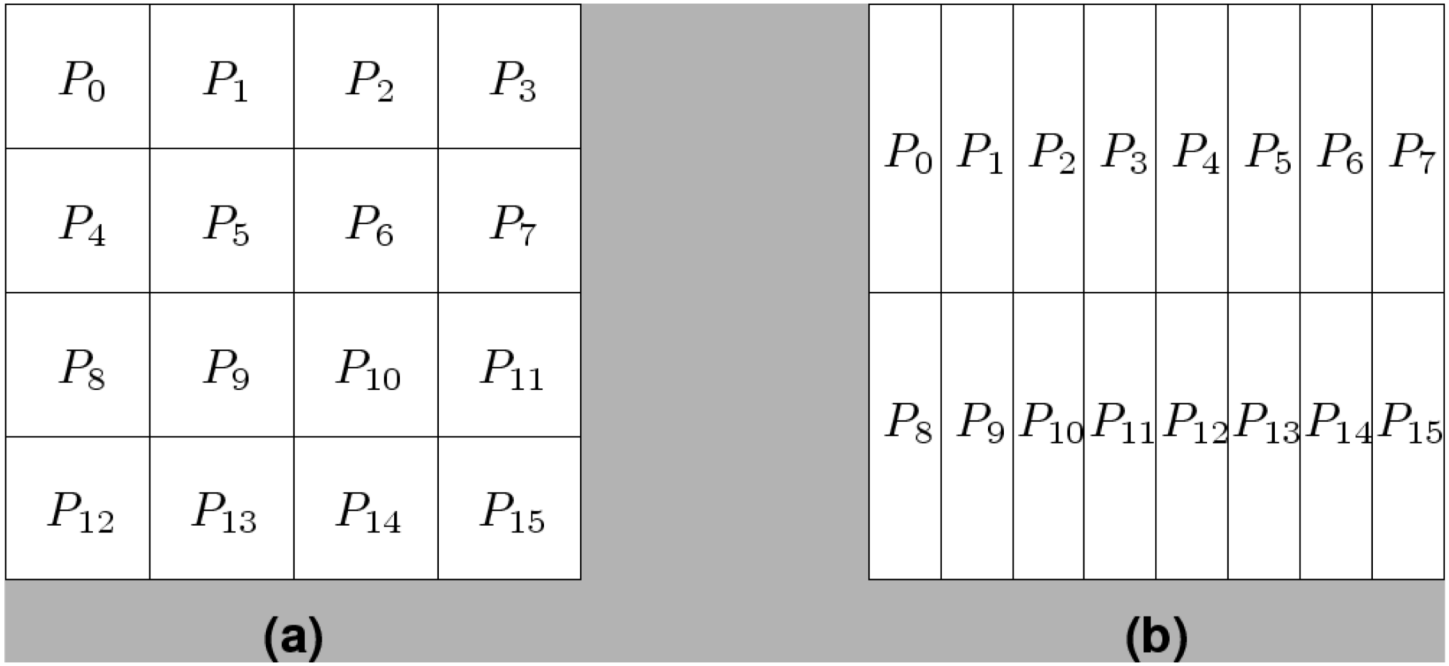
Global Vs Local Indexes

- In sequential code you always refer to global indexes
- With distributed data you must handle the distinction between global and local indexes (and possibly implementing utilities for transparent conversion)



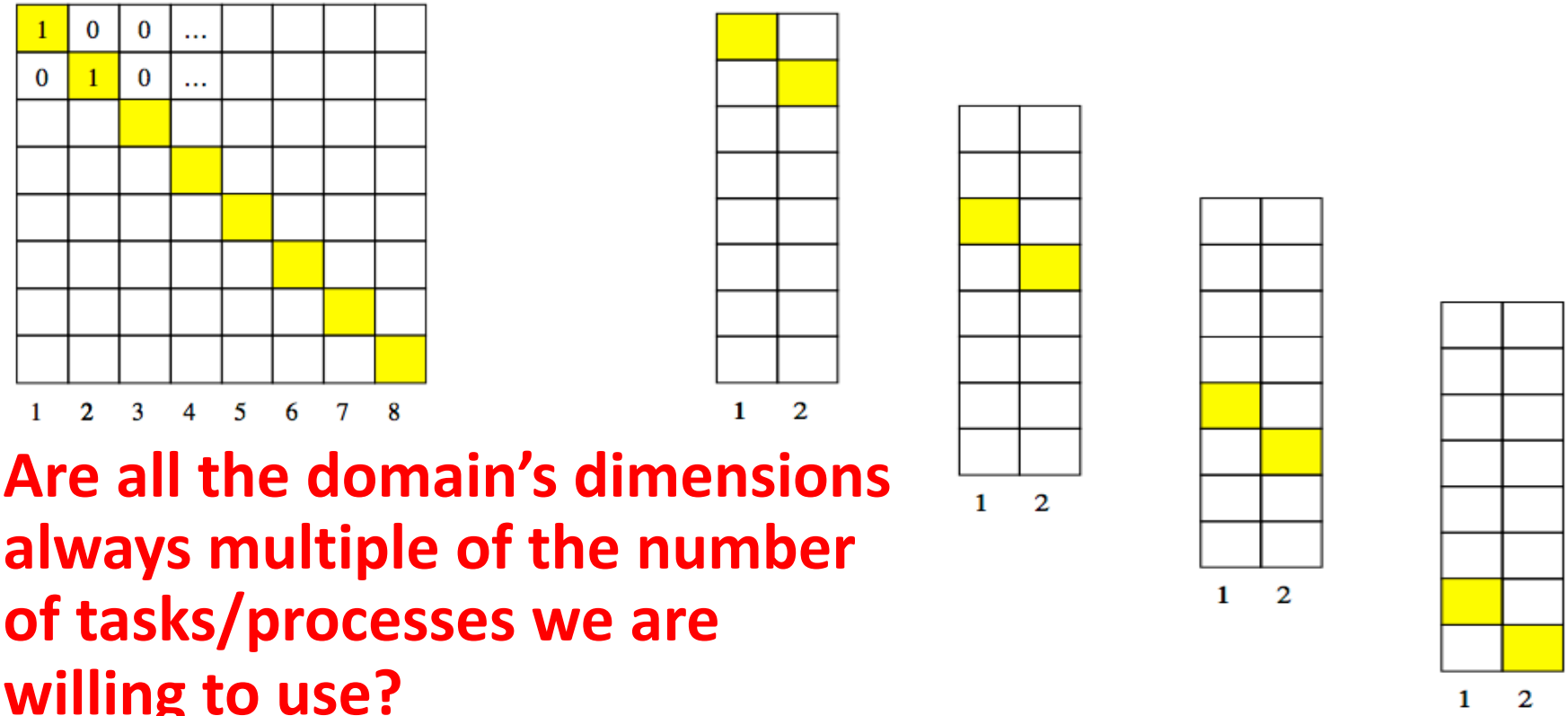
Block Array Distribution Schemes

Block distribution schemes can be generalized to higher dimensions as well.



Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution!!

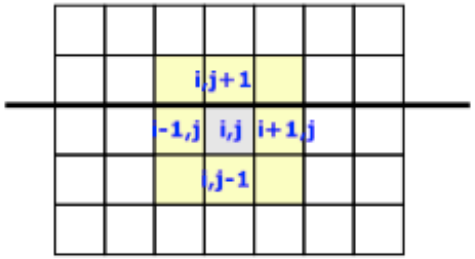
Collaterals to Domain Decomposition /1



Are all the domain's dimensions always multiple of the number of tasks/processes we are willing to use?

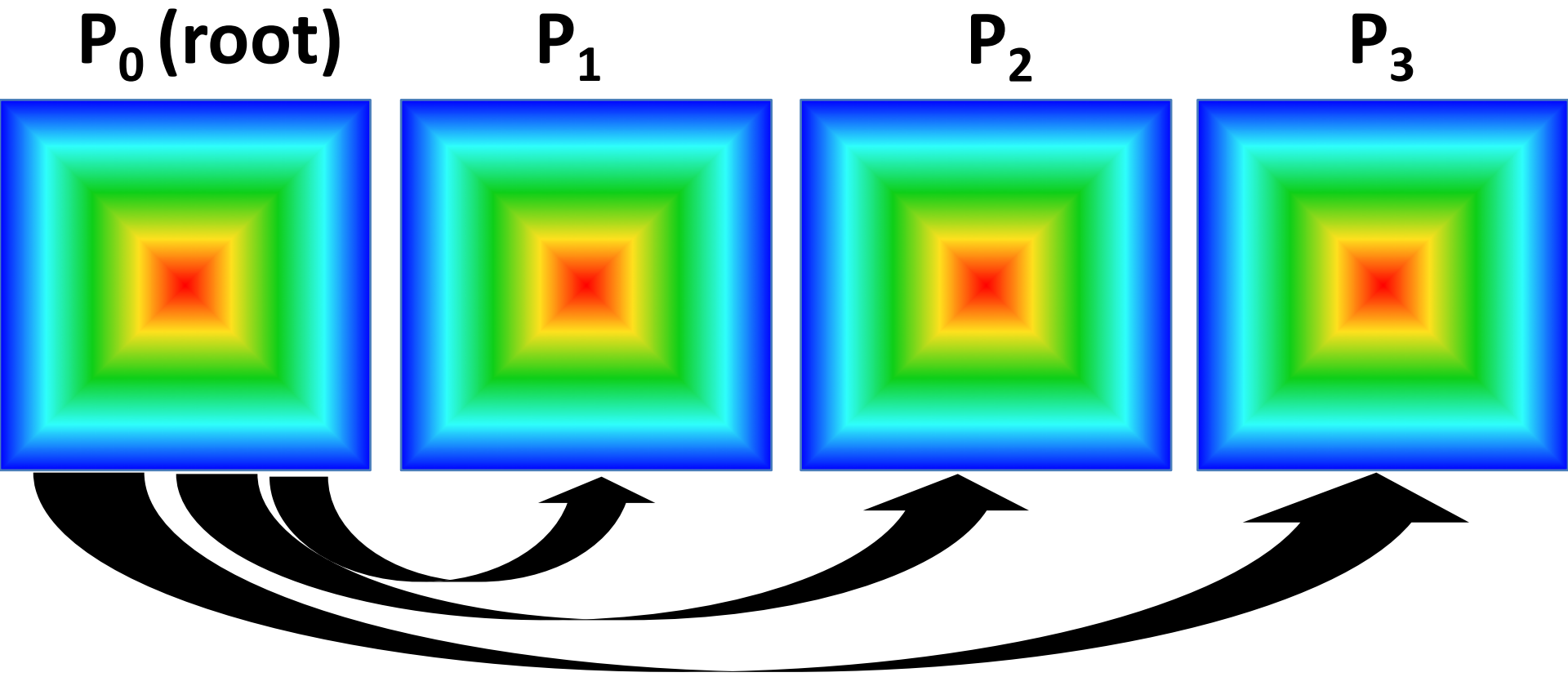
Again on Domain Decomposition

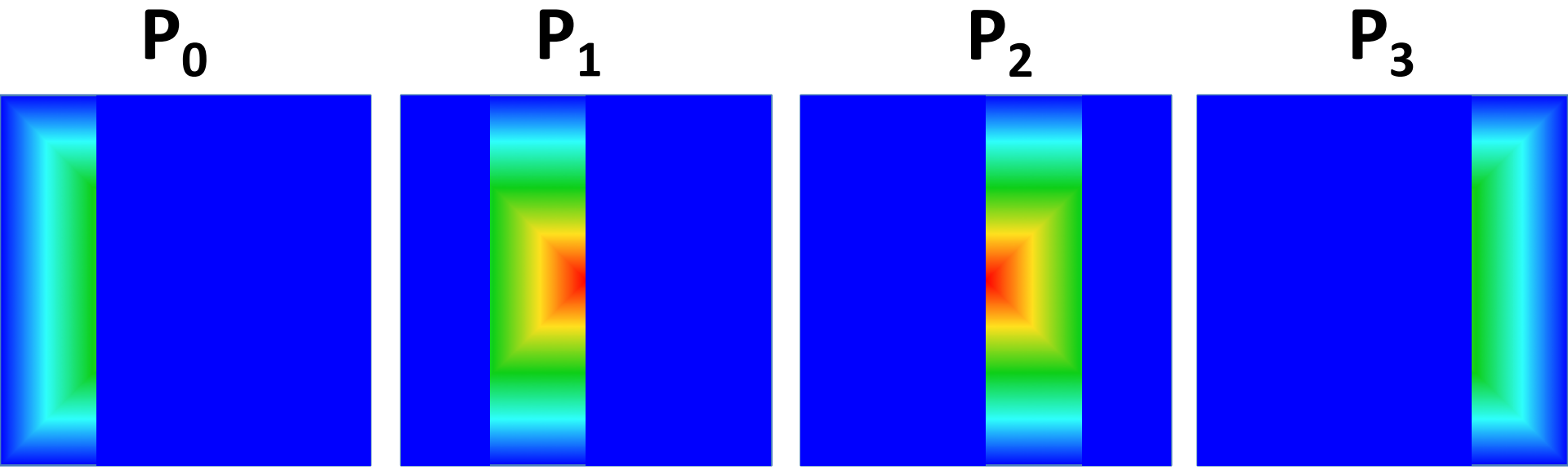
sub-domain boundaries



P_0

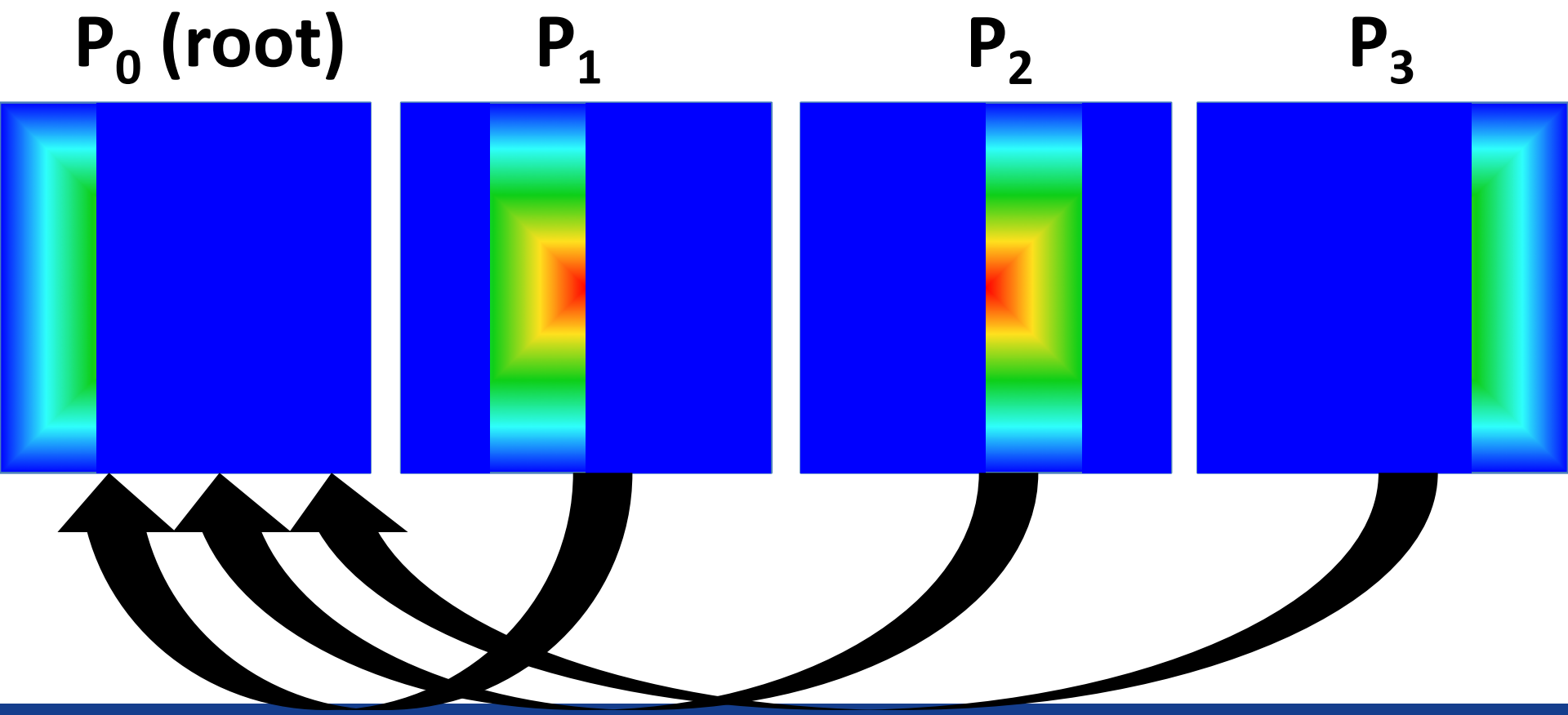
call MPI_BCAST(...)





call evolve(dtfact)

call MPI_Gather(…, …, …)

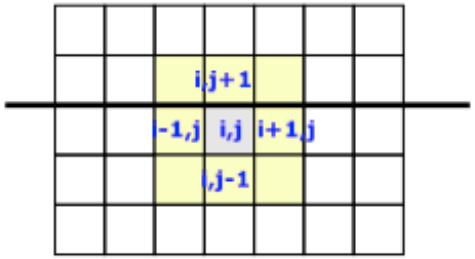


Replicated data

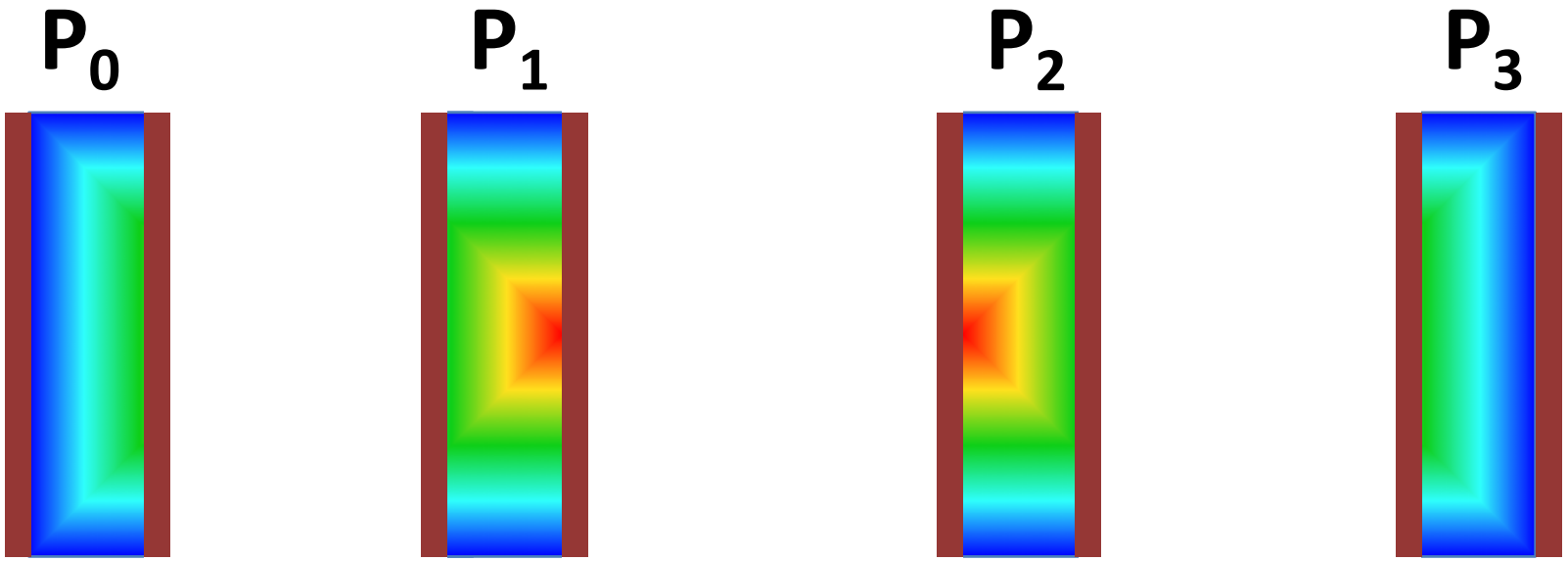
- Compute domain (and workload) distribution among processes
- Master-slaves: P_0 drives all processes
- Large amount of data communication
  - at each step P_0 distribute data to all processes and collect the contribution of each process
- Problem size scaling limited in memory capacity

Collaterals to Domain Decomposition /2

sub-domain boundaries

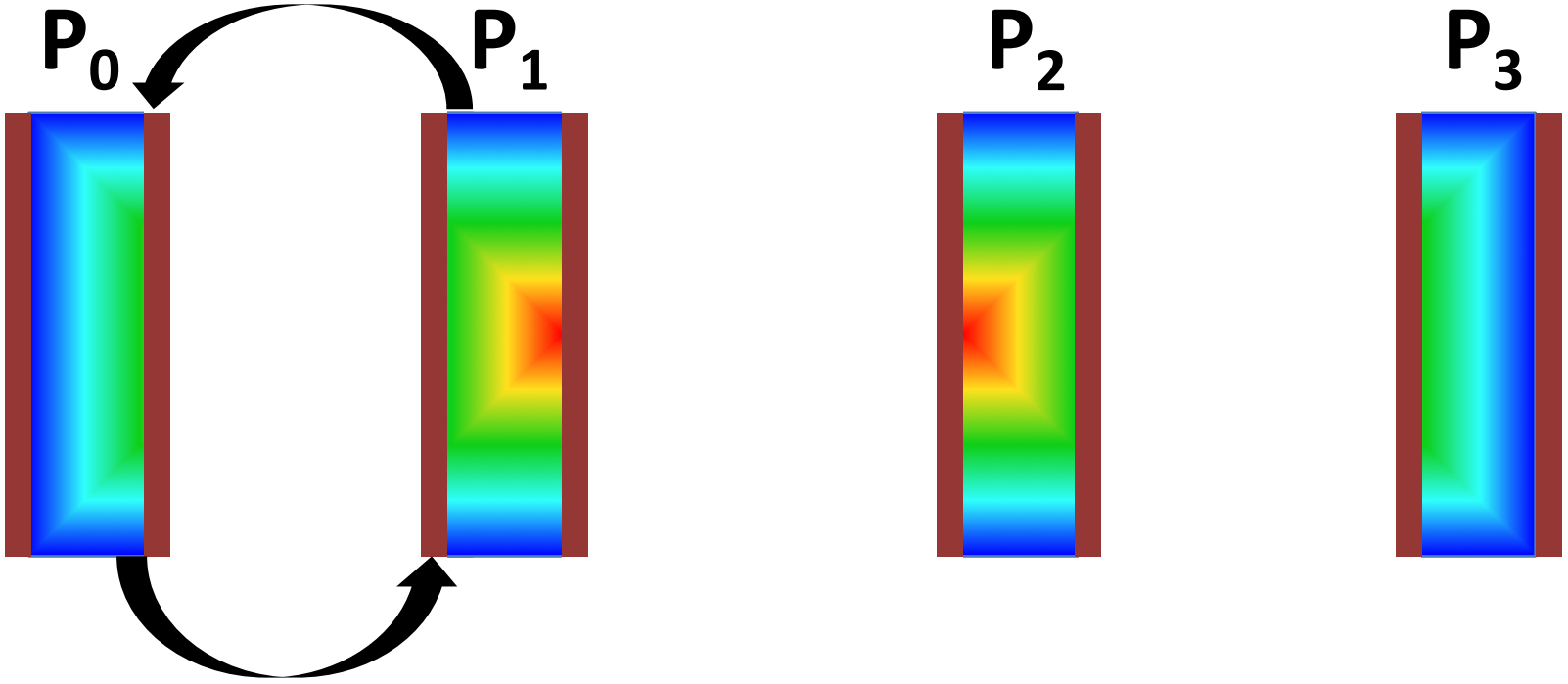


The Transport Code - Parallel Version

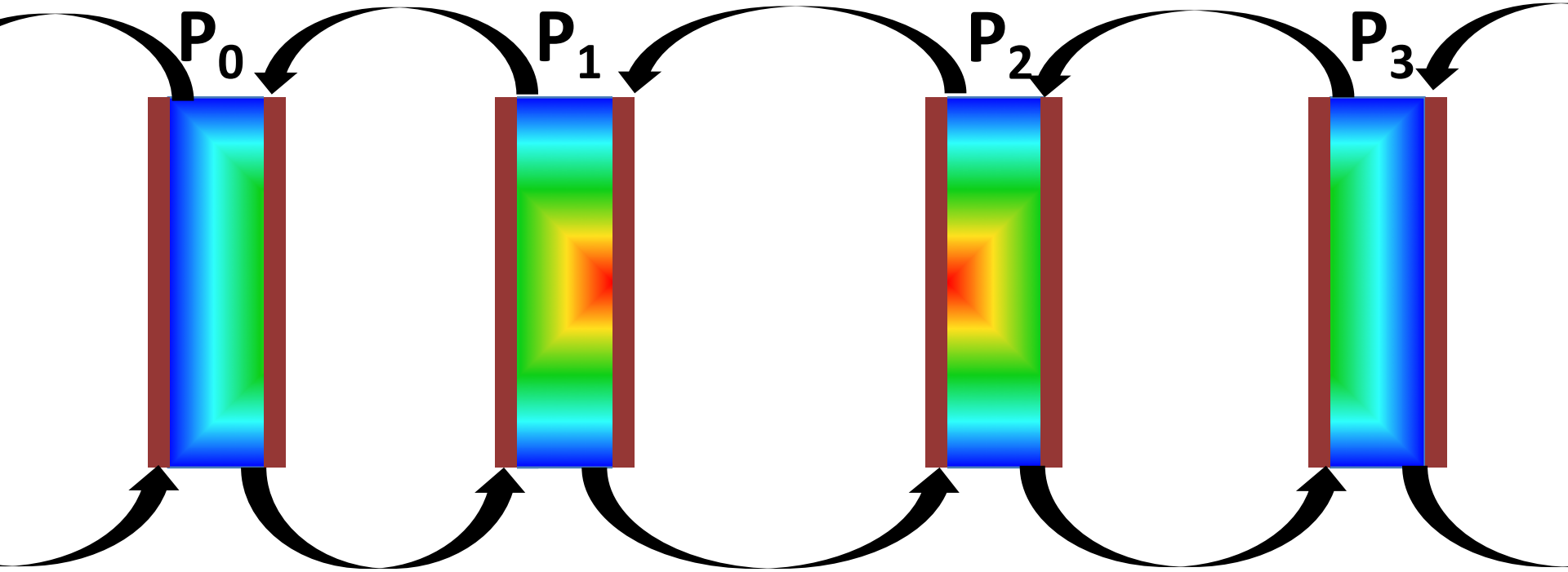


call evolve(dtfact)

Data exchange among processes



proc_down = mod(proc_me - 1 + nprocs , nprocs)

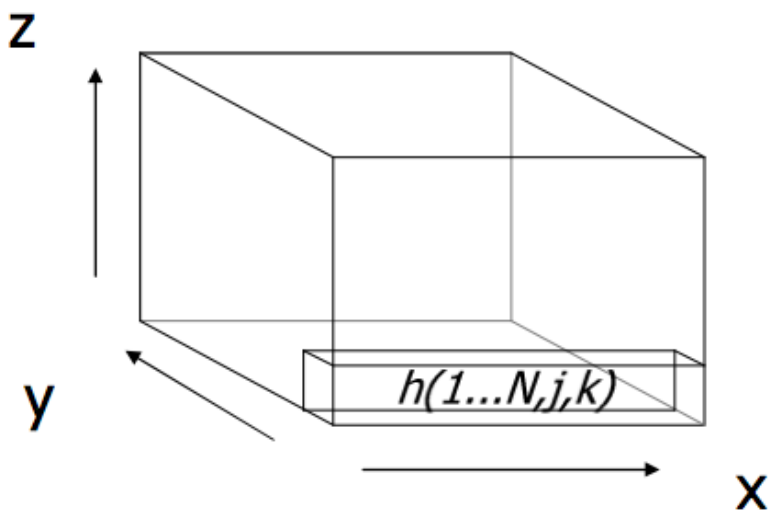


proc_up = mod(proc_me + 1 , nprocs)

Distributed Data

- Global and Local Indexes
- Ghost Cells Exchange Between Processes
  - Compute Neighbor Processes
- Parallel Output

Multidimensional FFT



$$f(x,y,z) = \frac{1}{N_z N_y N_x} \sum_{z=0}^{N_z-1} \Big(\sum_{y=0}^{N_y-1} \Big(\underbrace{\sum_{x=0}^{N_x-1} F(u,v,w)\, e^{-2\pi i \frac{xu}{N_x}}}_{\text{DFT long x-dimension}} \Big) e^{-2\pi i \frac{yv}{N_y}} \Big) e^{-2\pi i \frac{zw}{N_z}}$$

DFT long y-dimension

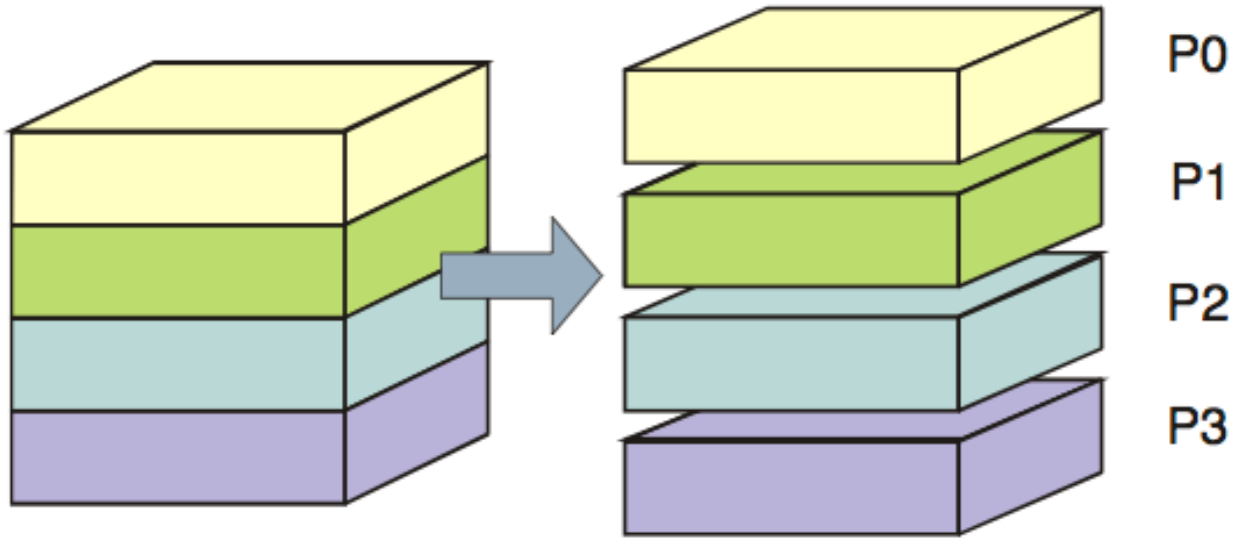
DFT long z-dimension

1) For any value of ***j*** and ***k*** transform the column (1…N, j, k)

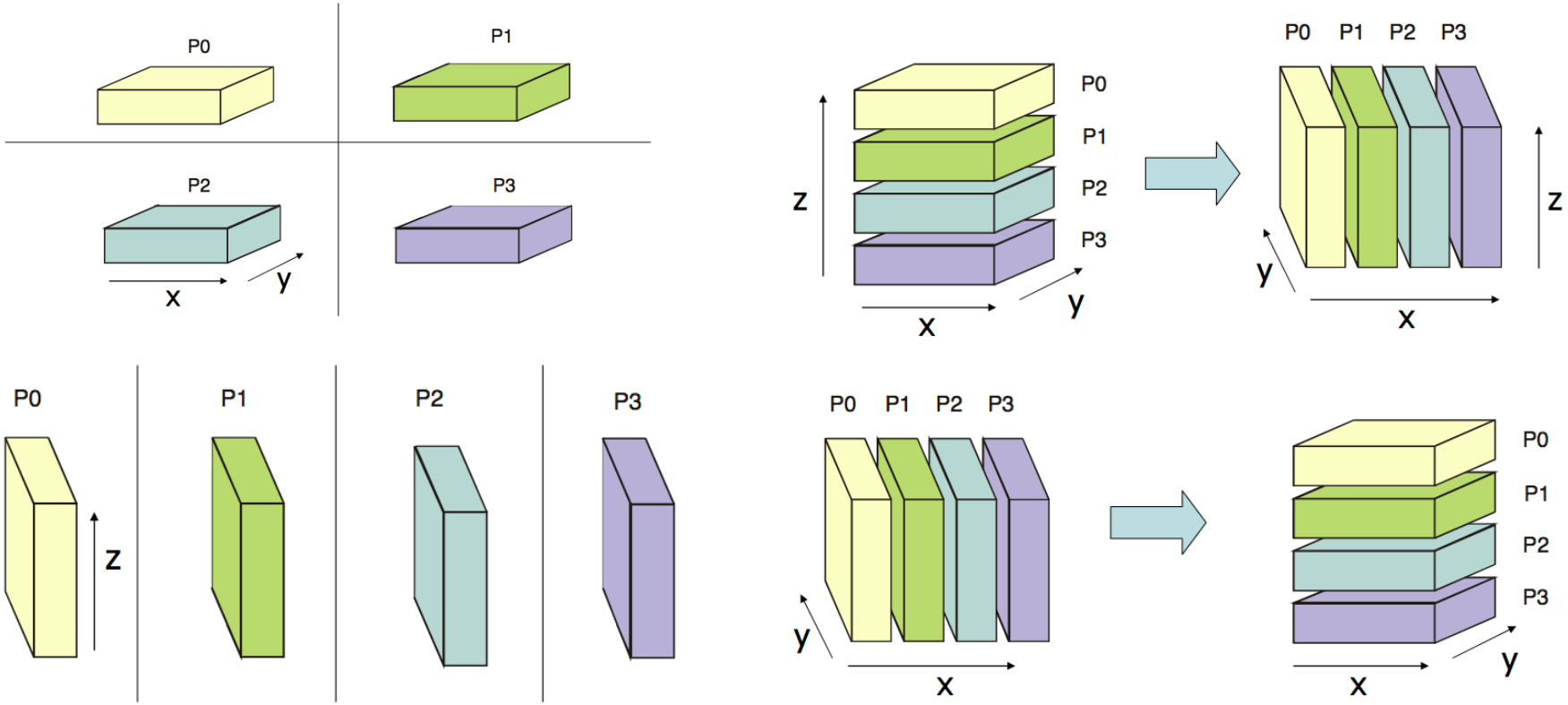
2) For any value of ***i*** and ***k*** transform the column (i, 1…N, k)

3) For any value of ***i*** and ***j*** transform the column (i, j, 1…N)

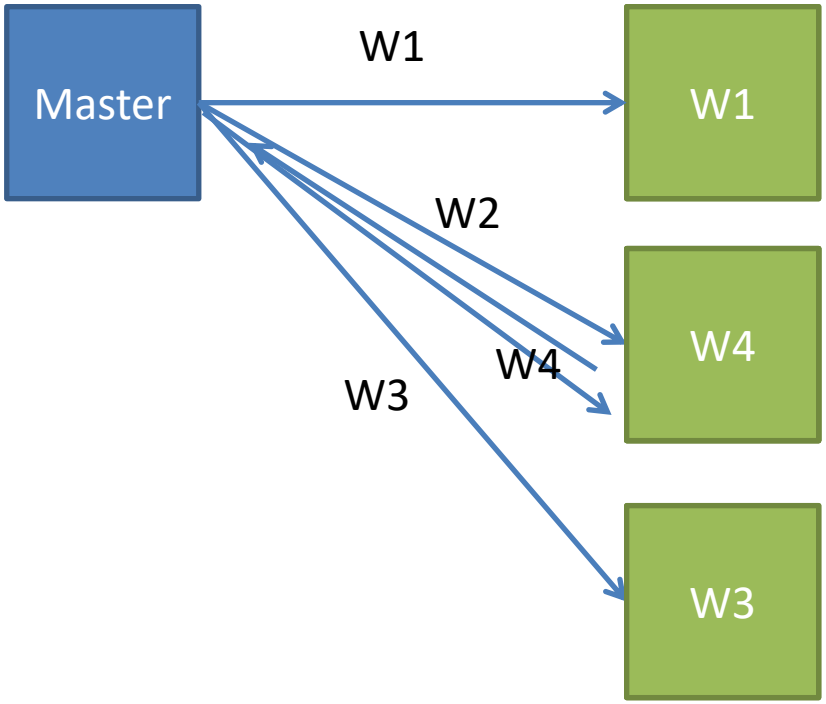
Parallel 3DFFT / 1



Parallel 3DFFT / 2



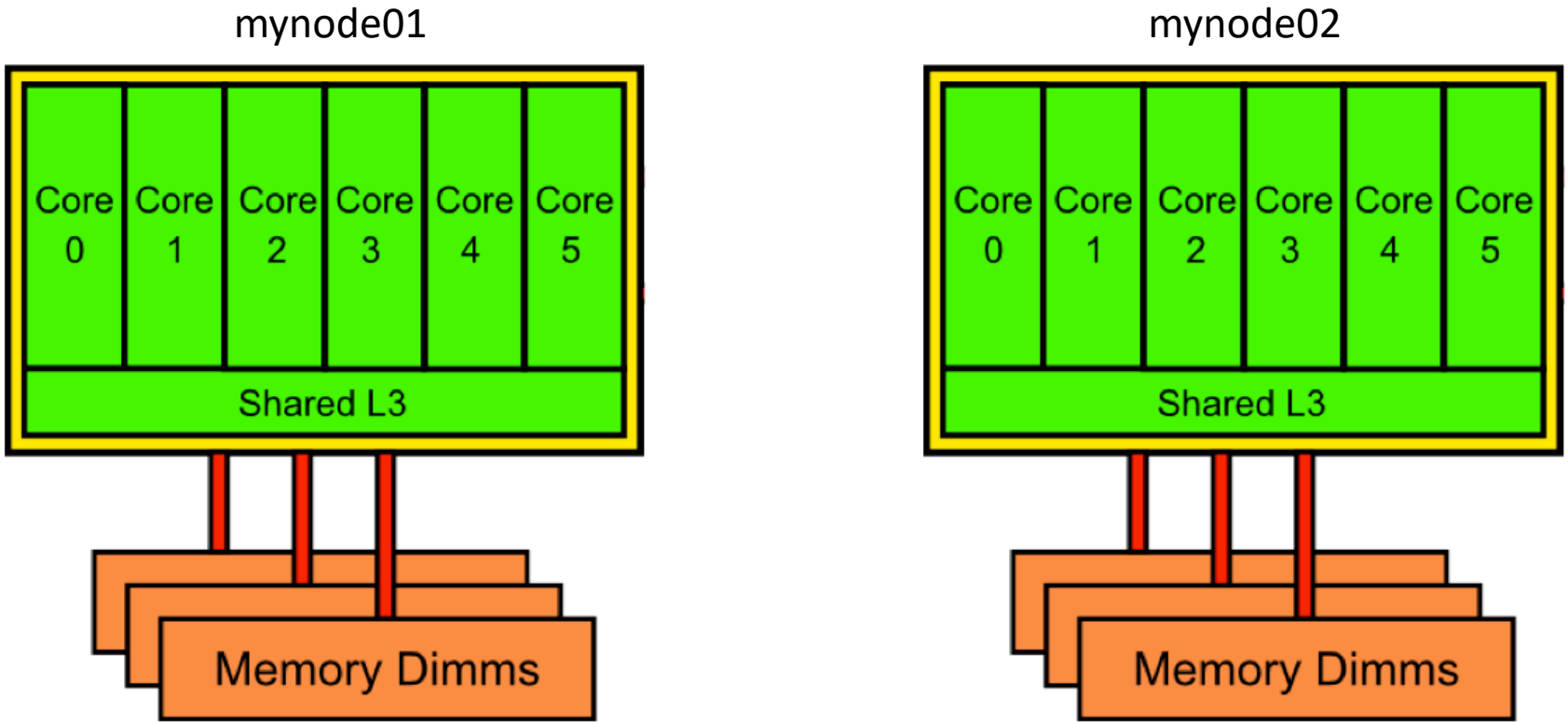
Master/Slave



Single Program on Multiple Data

- performing the same program (set of instructions) among different data
- Same model adopted by the MPI library
  - "yum install openmpi", "apt-get install openmpi", etc…
- A parallel tool is needed to handle the different processes working in parallel
- The MPI library provides the *mpirun* application to execute parallel instances of the same program

```
$ mpirun -np 12 my_program.x
```



```
[igirotto@mynode01 ~]$ mpirun -np 12 /bin/hostname
mynode01
mynode02
mynode01
mynode02
mynode01
mynode02
mynode01
mynode02
mynode01
mynode02
mynode01
mynode02
```

PATH name common to all processes !!

Parallel Operations in Practice

- Parallel reading and computing in parallel is always allowed
- Parallel writing is extremely dangerous!
- To control the parallel flow each process should be unique and identifiable (ID)
- The OpenMPI implementation of the MPI library provides a series of environment variables defined for each MPI process

OMPI_COMM_WORLD_SIZE - the number of processes in this process' MPI Comm_World

OMPI_COMM_WORLD_RANK - the MPI rank of this process

OMPI_COMM_WORLD_LOCAL_RANK - the relative rank of this process on this node within its job. For example, if four processes in a job share a node, they will each be given a local rank ranging from 0 to 3.

OMPI_UNIVERSE_SIZE - the number of process slots allocated to this job. Note that this may be different than the number of processes in the job.

OMPI_COMM_WORLD_LOCAL_SIZE - the number of ranks from this job that are running on this node.

OMPI_COMM_WORLD_NODE_RANK - the relative rank of this process on this node looking across ALL jobs.

http://www.open-mpi.org

In Python

```
import os
myid = os.environ['OMPI_COMM_WORLD_RANK']
[...]
```

In BASH

```
#!/bin/bash
myid=${OMPI_COMM_WORLD_RANK}
[...]
```

```
[igirotto@mynode01 ~]$ mpirun ./myprogram.[py/sh...]
```

Possible Applications

- Executing multiple instances on the same program with different inputs/initial cond.
- Reading large binary files by splitting the workload among processes
- Searching elements on large data-sets
- Other parallel execution of embarrassingly parallel problem (no communication among tasks)

Summary /2

- Task Farming is a simple model to parallelize simple problems that can be divided in independent task
- The *mpirun* application aids to easily perform multiple processes, includes environment setting
- Load balancing remains a main problem, but moving from serial to parallel processing can substantially speed-up time of simulation

Task Farming

- Many independent programs (tasks) running at once
  - each task can be serial or parallel
  - "independent" means they don't communicate directly
  - Processes possibly driven by the mpirun framework

```
[igirotto@localhost]$ more my_shell_wrapper.sh
#!/bin/bash
#example for the OpenMPI implementation
./prog.x --input input_${OMPI_COMM_WORLD_RANK}.dat

[igirotto@localhost]$ mpirun -np 400 ./my_shell_wrapper.sh
```

Easy Parallel Computing

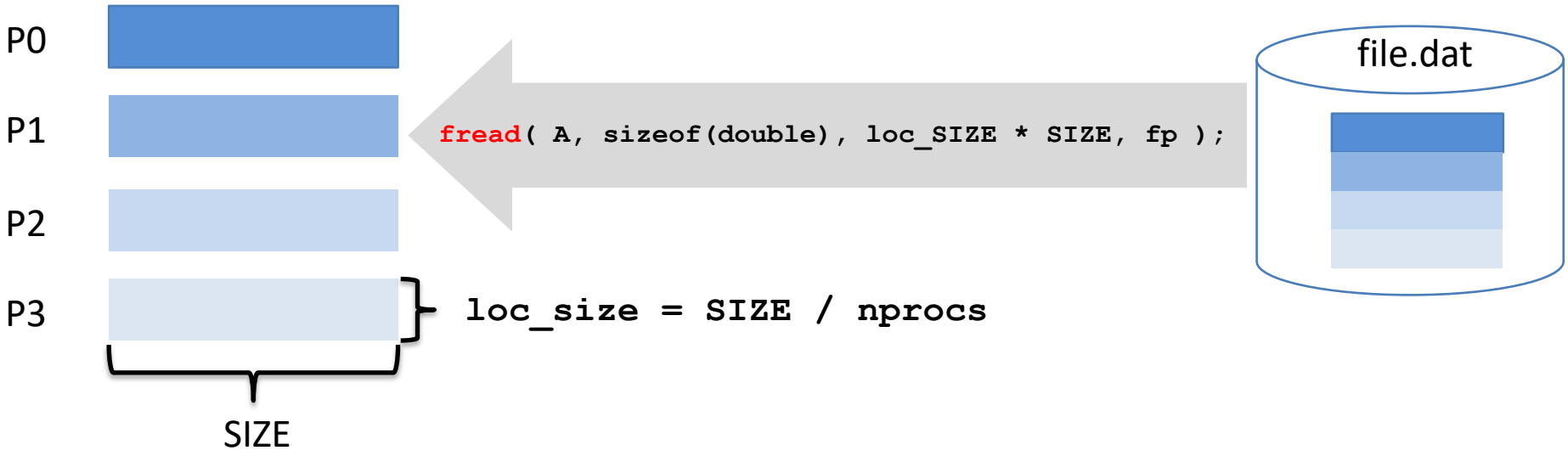
- Farming, embarrassingly parallel
  - Executing multiple instances on the same program with different inputs/initial cond.
  - Reading large binary files by splitting the workload among processes
  - Searching elements on large data-sets
  - Other parallel execution of embarrassingly parallel problem (no communication among tasks)
- Ensemble simulations (weather forecast)
- Parameter space (find the best wing shape)

How to read a FILE in parallel?

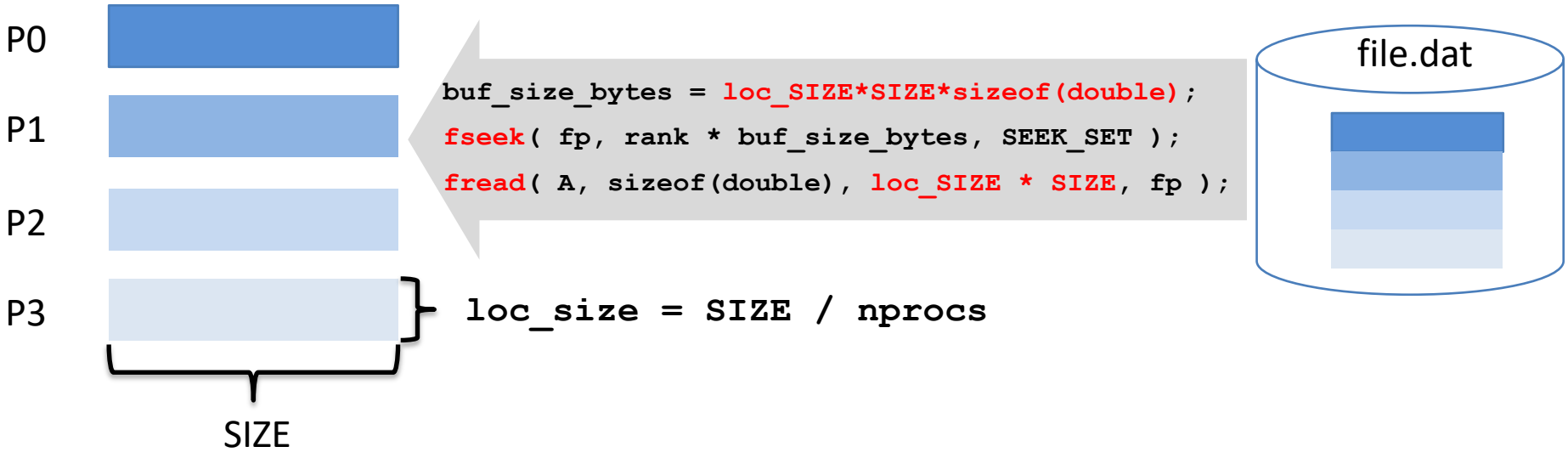
- It requires a serialized access from one or more processes in case of a text file (sequential access)
- It can be performed in parallel in case of binary files (random access)
- Common sequence for creating and writing a binary file:

```
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 distributed data on a FILE?

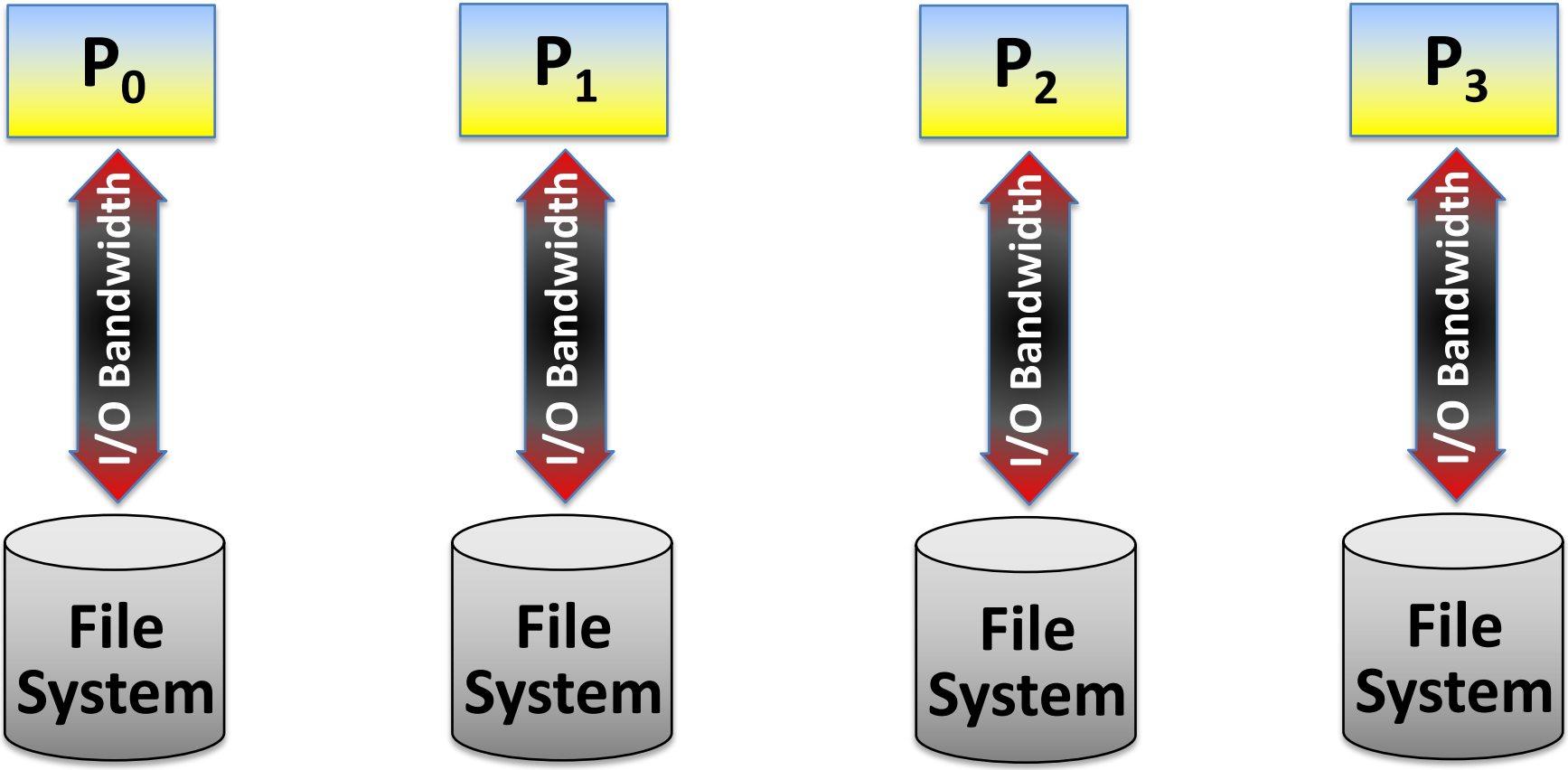


How to read distributed data on a FILE?

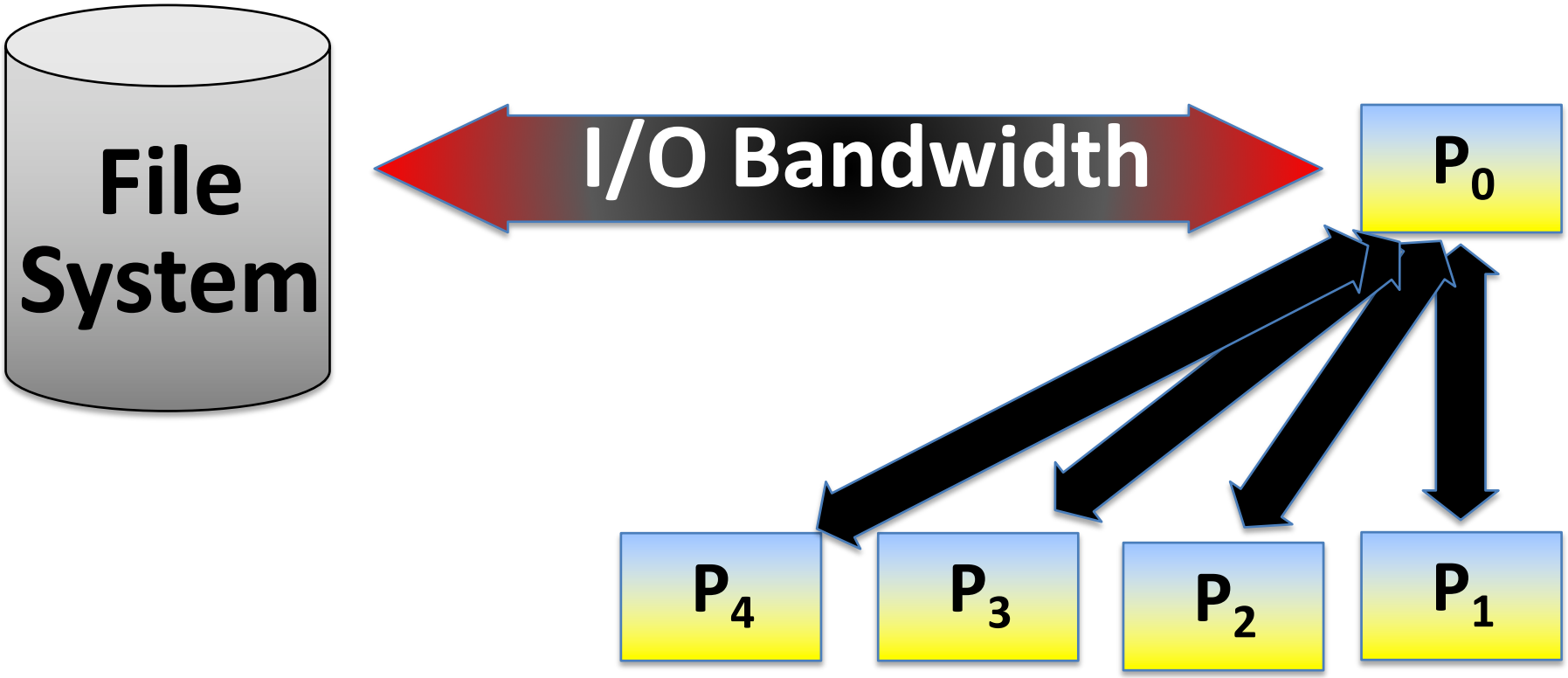


The fseek() function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file,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

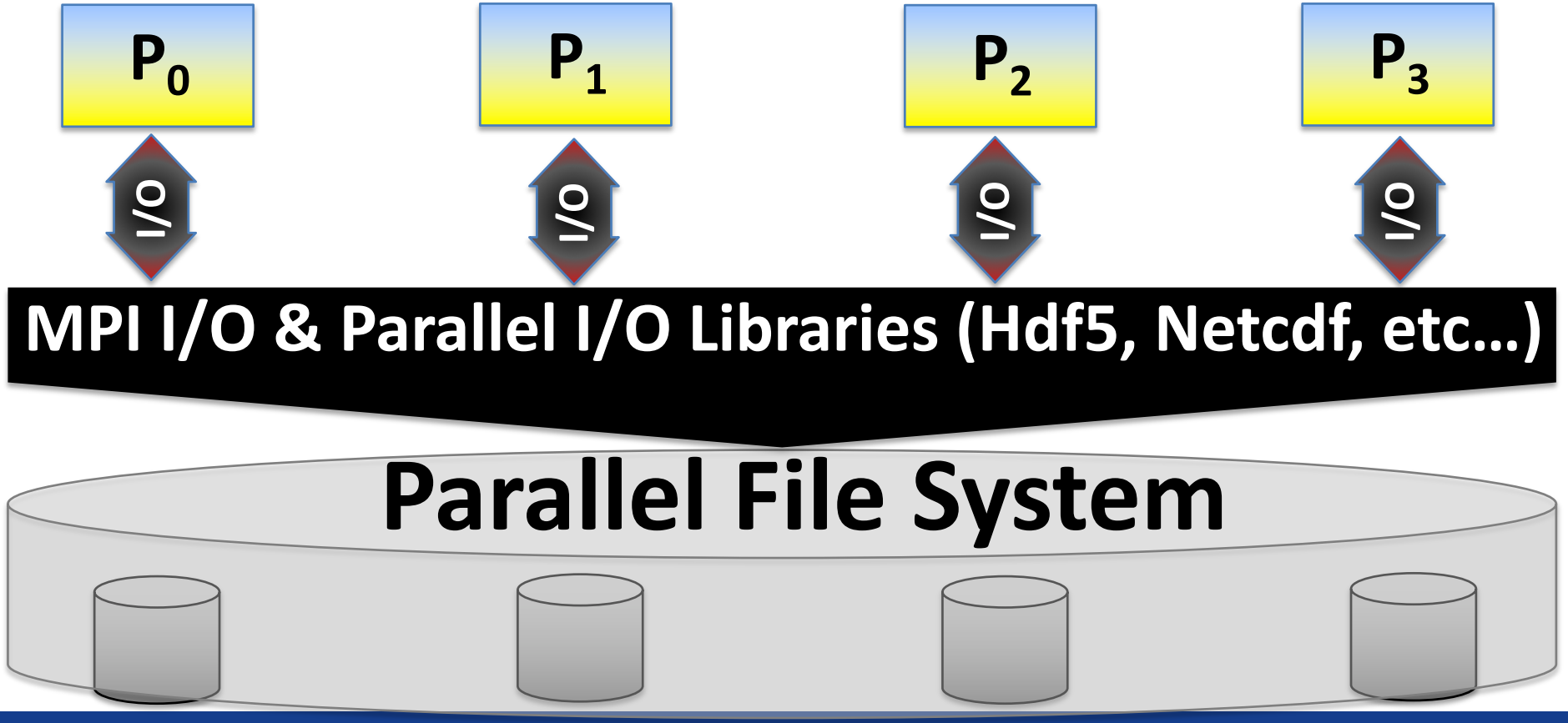
Parallel I/O


P0
P1
P2
P3
I/O Bandwidth
I/O Bandwidth
I/O Bandwidth
I/O Bandwidth
File System
File System
File System
File System

Parallel I/O



Parallel I/O


P0
P1
P2
P3
I/O
I/O
I/O
I/O
MPI I/O & Parallel I/O Libraries (Hdf5, Netcdf, etc…)
Parallel File System

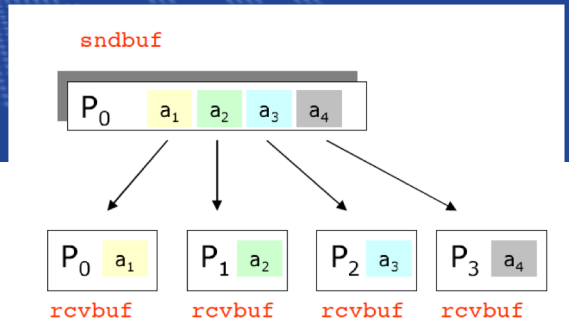
Make Use Freely Available Parallel Libraries

- Scalable Parallel Random Number Generators Library (SPRNG)
- Parallel Linear Algebra (ScaLAPACK)
- Parallel Library for Solution of Finite Elements (dealii)
- Parallel Library for FFT (FFTW)
- Parallel Linear Solver for Sparce Matrices (PETSc)

Collective Communications

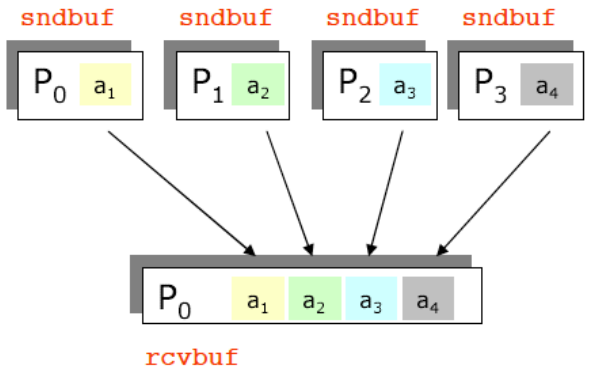
- At the bottom line are based on point 2 point (you could build your own)
- The MPI include a series of subroutines to handle different patterns of communications: 1 to N, N to 1 and N to N
- **Collective Communications imply a *synchronization point* among processes**

MPI_Scatter



One-to-all communication: different data sent from root process to all others in the communicator.

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
[IN sendbuf] starting address of send buffer (choice)
[IN sendcount] number of elements sent to each process (integer, significant only at root) [IN sendtype] data type of send buffer elements (significant only at root) (handle)
[OUT recvbuf] address of receive buffer (choice)
[IN recvcount] number of elements in receive buffer (integer)
[IN recvtype] data type of recv buffer elements (handle)
[IN root] rank of receiving process (integer)
[IN comm] communicator (handle)



MPI_Gather

One-to-all communication: different data collected by the root process, from all others processes in the communicator.

It is the opposite of Scatter

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
[IN sendbuf] starting address of send buffer (choice)
[IN sendcount] number of elements in send buffer (integer)
[IN sendtype] data type of send buffer elements (handle)
[OUT recvbuf] address of receive buffer (choice, significant only at root)
[IN recvcount] number of elements for any single receive (integer, significant only at root)
[IN recvtype] data type of recv buffer elements (significant only at root) (handle)
[IN root] rank of receiving process (integer)
[IN comm] communicator (handle)

STANDARD BLOCKING SEND - RECV

MPI_SEND(buf, count, type, dest, tag, comm, ierr)

MPI_RECV(buf, count, type, dest, tag, comm, status, ierr)

Buf array of MPI type **type**.

Count (INTEGER) number of element of **buf** to be sent/recv

Type (INTEGER) MPI type of **buf**

Dest (INTEGER) rank of the destination process

Tag (INTEGER) number identifying the message

Comm (INTEGER) communicator of the sender and receiver

*** Status** (INTEGER) array of size **MPI_STATUS_SIZE** containing communication status information

Ierr (INTEGER) error code

**** used only for receive operations***

NON-BLOCKING SEND - RECV

MPI_ISEND(buf, count, type, dest, tag, comm, request, ierr)

MPI_IRECV(buf, count, type, dest, tag, comm, request, ierr)

Buf array of MPI type **type**.

Count (INTEGER) number of element of **buf** to be sent/recv

Type (INTEGER) MPI type of **buf**

Dest (INTEGER) rank of the destination process

Tag (INTEGER) number identifying the message

Comm (INTEGER) communicator of the sender and receiver

Request (INTEGER) request handler, used for checking the communication status

Ierr (INTEGER) error code

No-Blocking Checkpoint

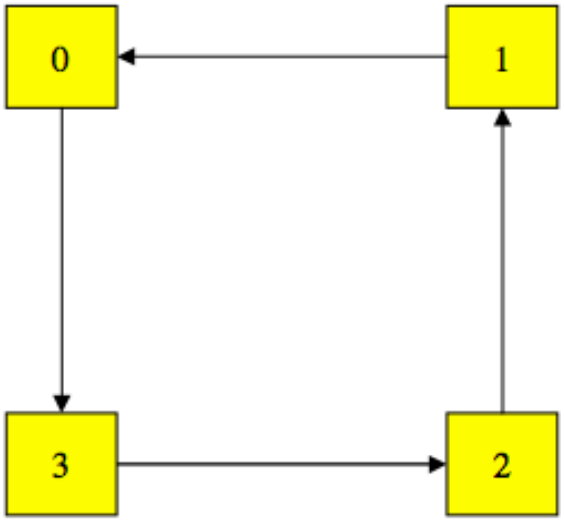
MPI_WAIT(request, status, ierr)

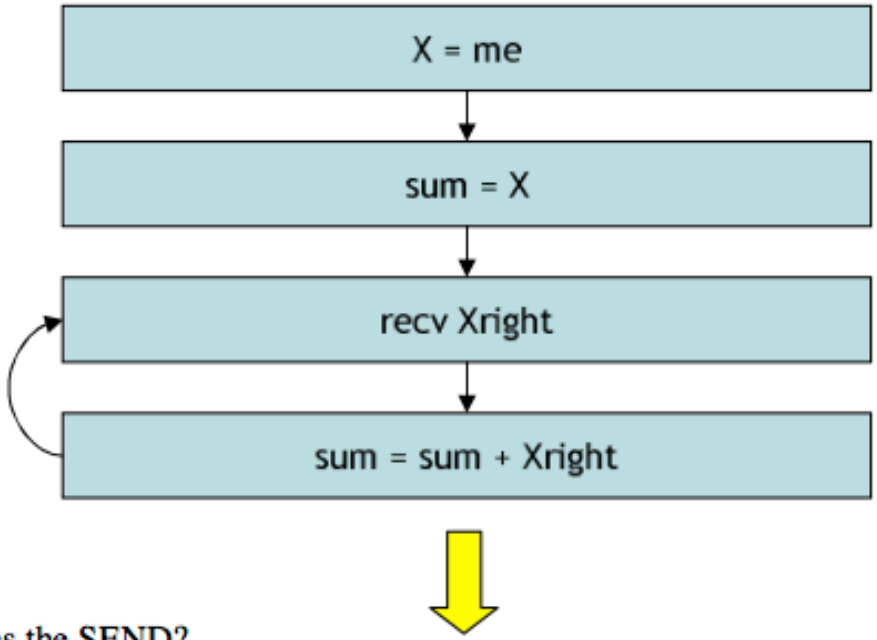
Request (INTEGER) request handler, used for checking the communication status

Status (INTEGER) array of size **MPI_STATUS_SIZE** containing communication status information

Ierr (INTEGER) error code

Wait until the communication handled by the object request is terminated. For test only use MPI_TEST, for checkpoint of many communication use MPI_WAITALL





Who/when does the SEND?

What does it send?

How many times?

sum = 6 (on 4 processors)

Exercise (2)

- Implement the proposed exercise, first exchanging one single element (mype) among processes as illustrated in class as well as on the previous slide. Try to optimize the code for sending in the ring a large set of data and overlapping the computation (Σ) and the communication (send-recv). In case of a dataset larger than one element the local sum is considered a vector sum (element by element).

Exercise (1)

- Implement a code to initialize a distributed identity matrix of size (N,N). Print the matrix ordered on standard output if N is smaller than 10, otherwise on a binary file.
- (Plus) Implement I/O overlapping the receiving data on process 0 with no-blocking communication, overlapping communication and I/O.