



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)

a program should operate on data that is too big to be handled by a single process: it'd require too much memory. So u divide data domain in slices and the



Static Data Partitioning

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

distribute between processes the less nested index (C rowwise->fastest is A[ro]

row-wise distribution

P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7

column-wise distribution

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
-------	-------	-------	-------	-------	-------	-------	-------



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

FIRST THING U DO: create a function that maps global index->local index

- 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)

Local Idx

1	2	3
---	---	---

1	2	3
---	---	---

1	2	3
---	---	---

Global Idx

1	2	3
---	---	---

4	5	6
---	---	---

7	8	9
---	---	---

Block Array Distribution Schemes

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

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

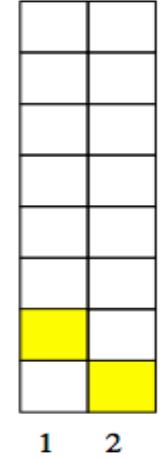
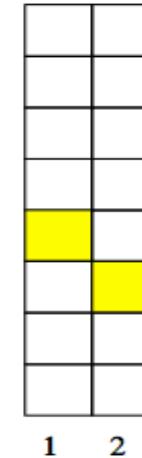
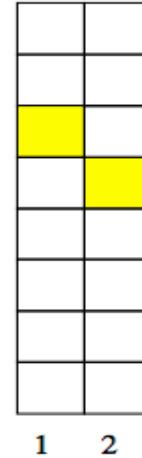
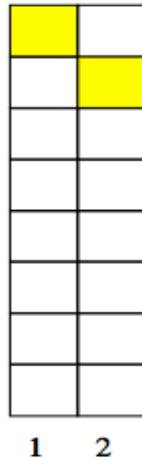
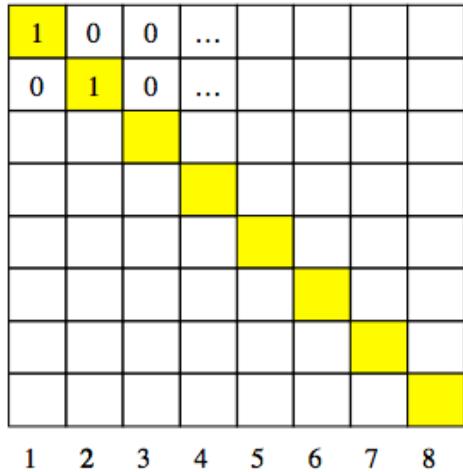
(a)

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

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

Collaterals to Domain Decomposition /1

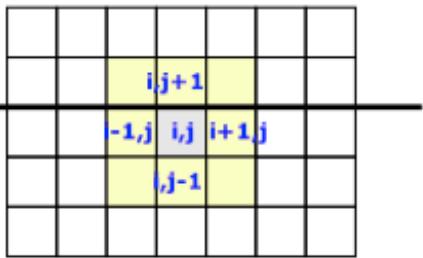


Are all the domain's dimensions always multiple of the number of tasks/processes we are willing to use?



Again on Domain Decomposition

sub-domain boundaries



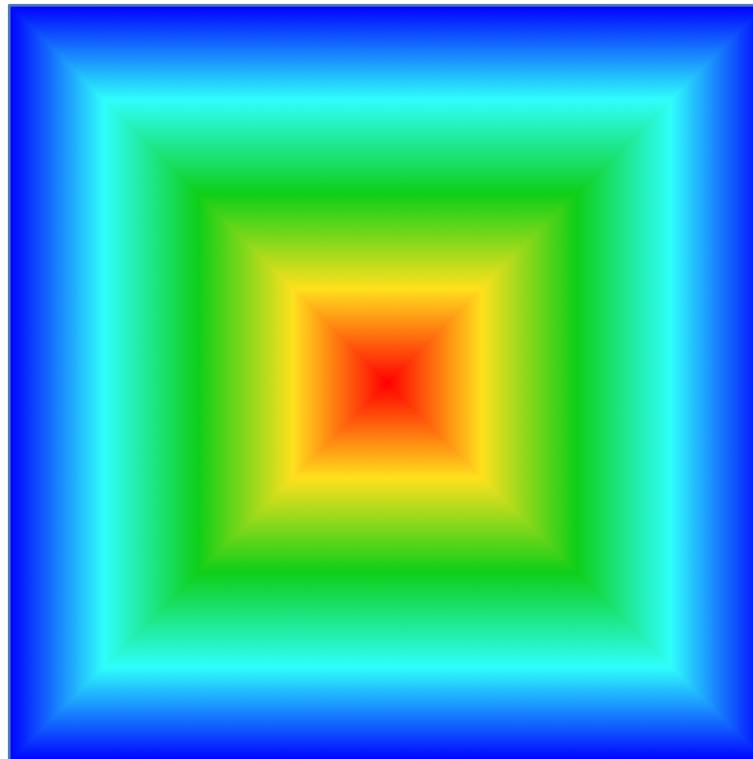
usually to compute data in a point u need data in the neighbourhood, so what if I need data in another proc? I c



The Abdus Salam
International Centre
for Theoretical Physics



P_0





The Abdus Salam
International Centre
for Theoretical Physics



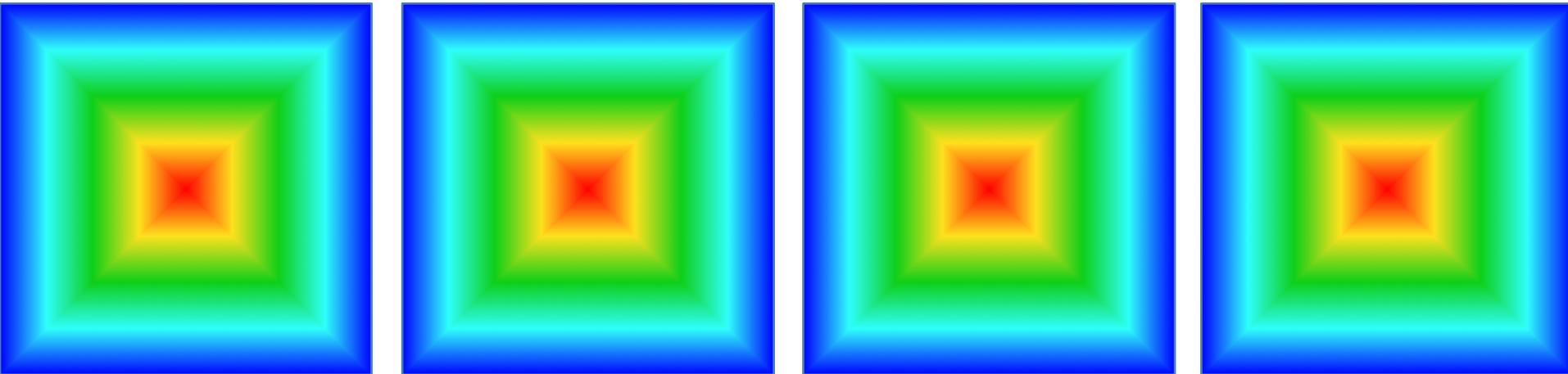
call MPI_BCAST(...)

P_0 (root)

P_1

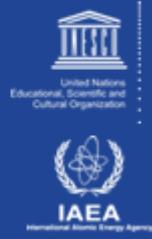
P_2

P_3





The Abdus Salam
International Centre
for Theoretical Physics



P_0

P_1

P_2

P_3



call evolve(dtfact)

process 0 write on file and fill until line 9 included. To write in the same file the data of P1, u send all the data of P1 to P0 and write that into the same file in P

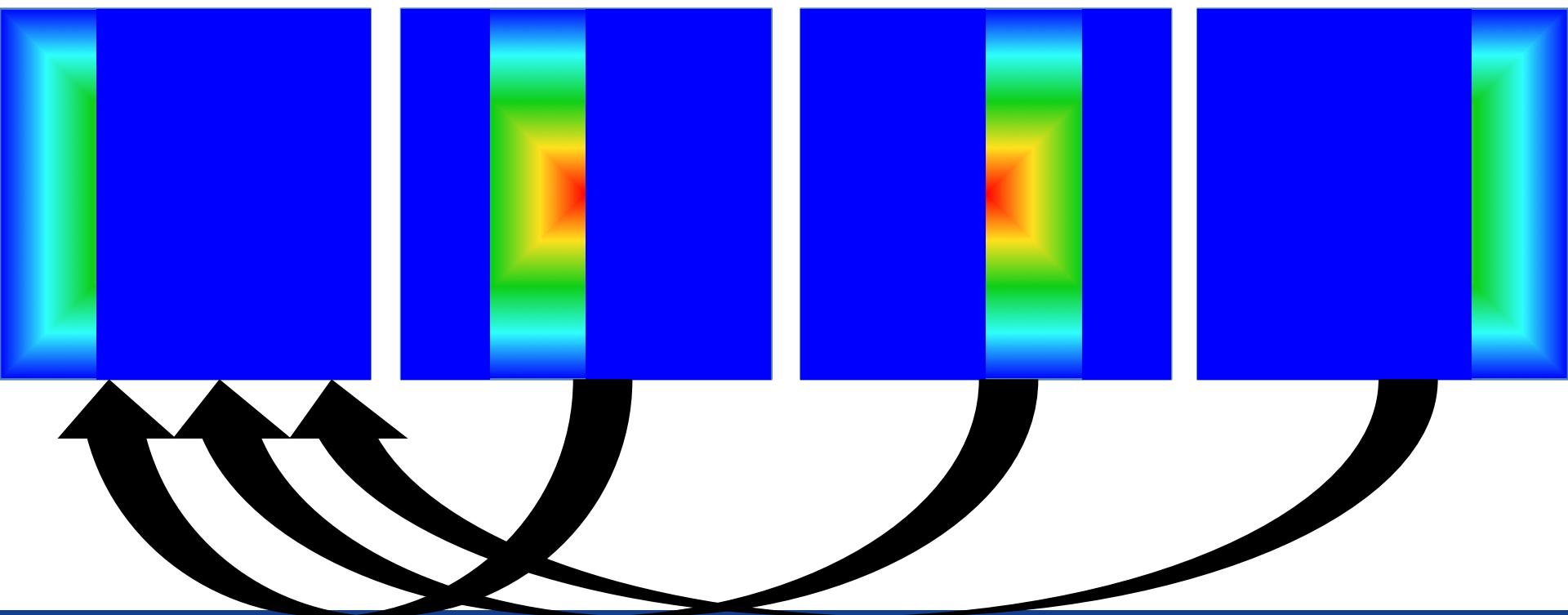
call MPI_Gather(..., ..., ...)

P₀ (root)

P₁

P₂

P₃





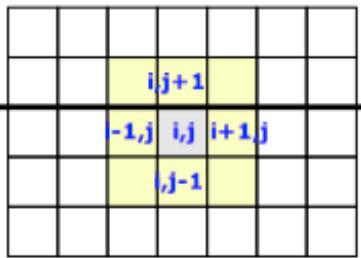
Replicated data

- 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

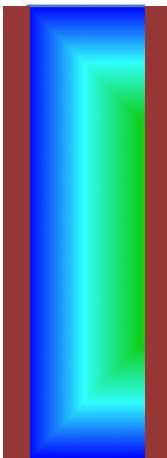


ghost cell approach: REduce from 1:N communication to point to point: u just keep sending one line of the matrix

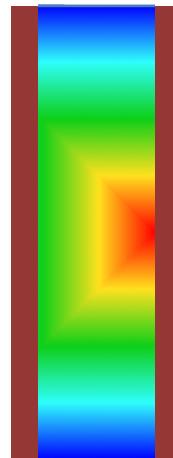


The Transport Code - Parallel Version

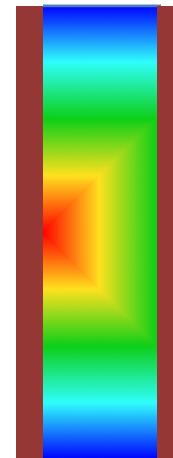
P_0



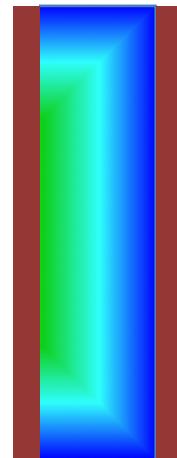
P_1



P_2

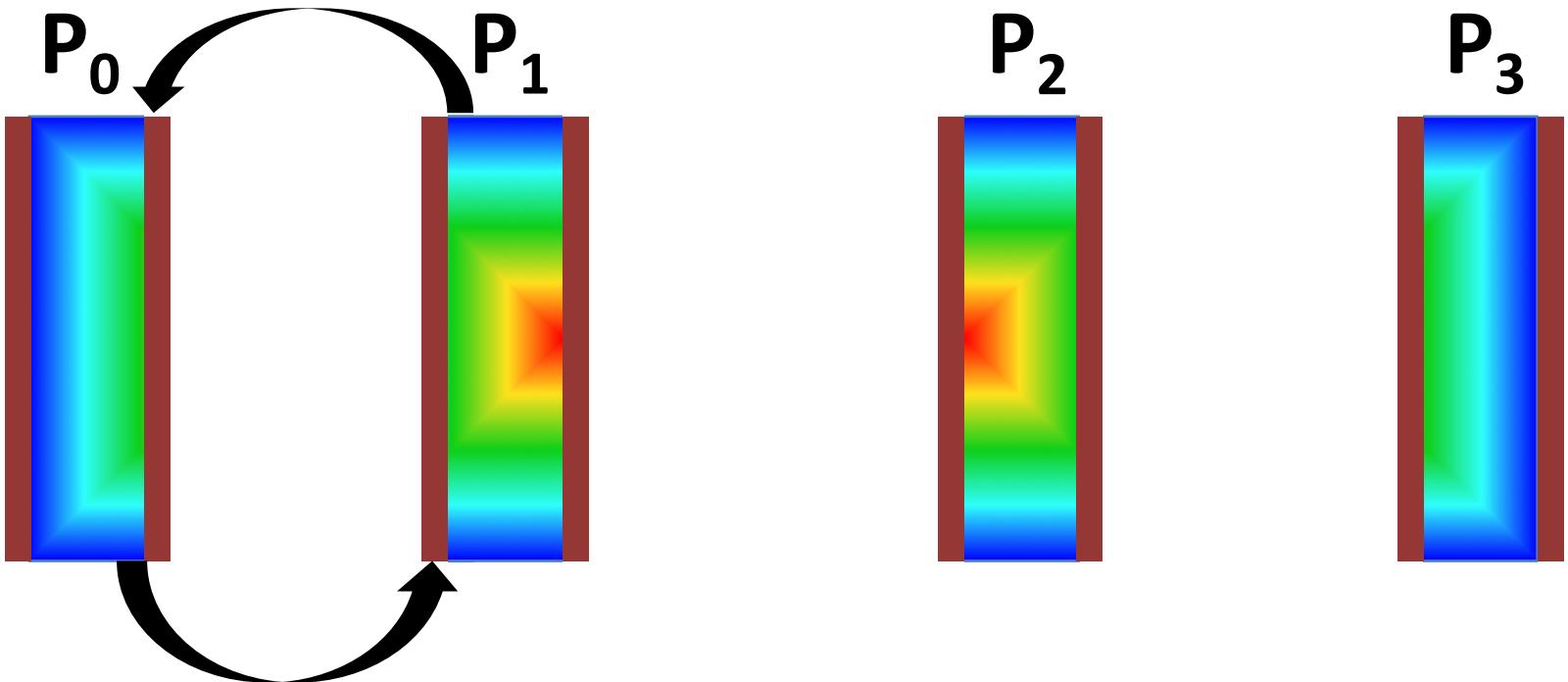


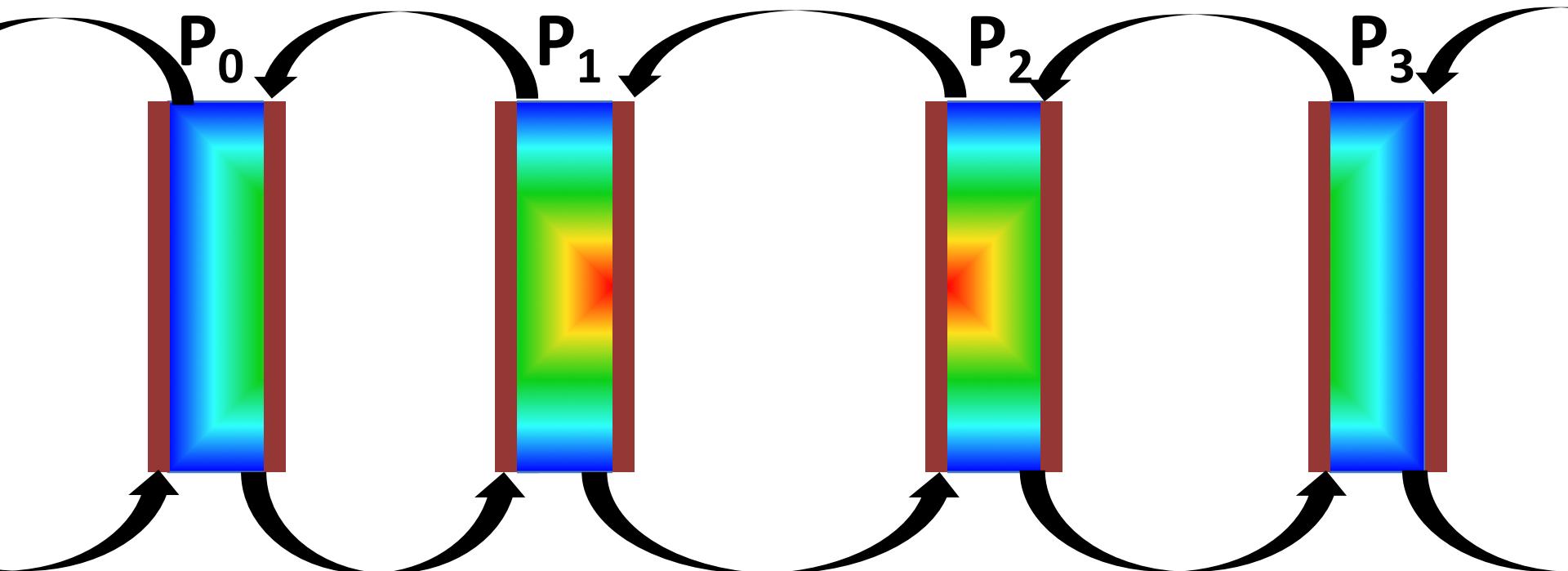
P_3



call evolve(dtfact)

Data exchange among processes



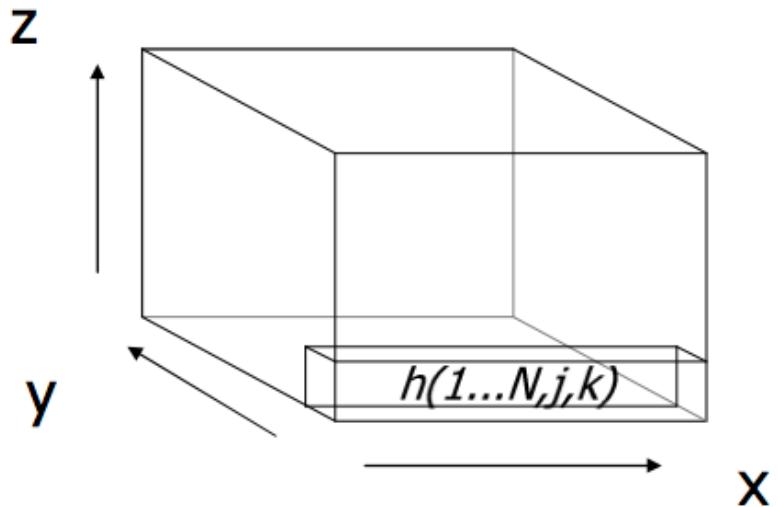
$$\text{proc_down} = \text{mod}(\text{proc_me} - 1 + \text{nprocs}, \text{nprocs})$$

$$\text{proc_up} = \text{mod}(\text{proc_me} + 1, \text{nprocs})$$



Distributed Data

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

Multidimensional FFT

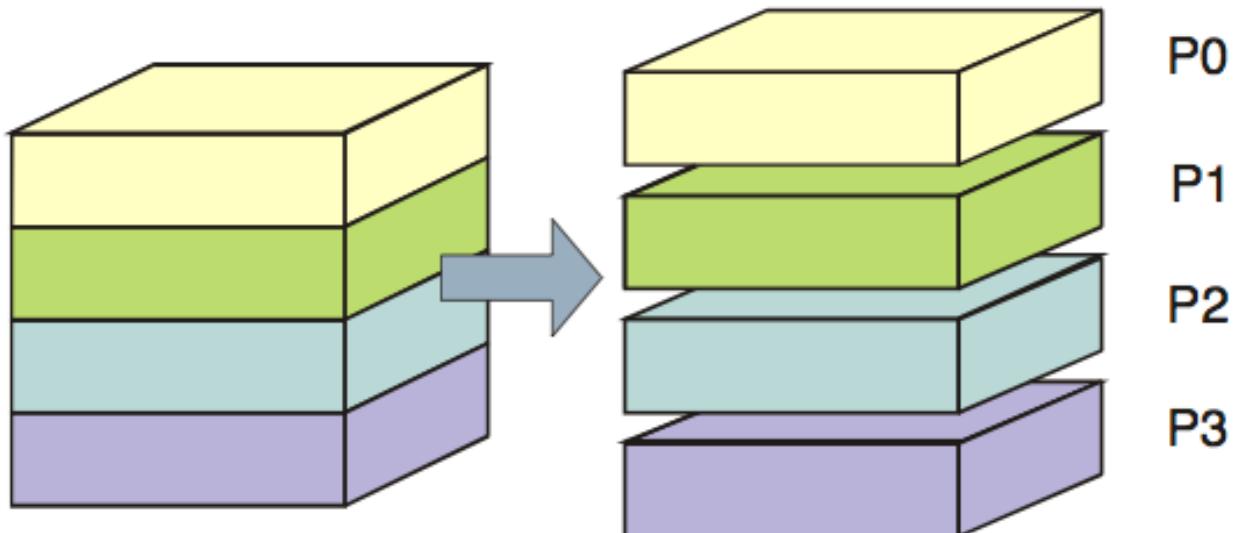


- 1) For any value of j and k transform the column $(1\dots N, j, k)$
- 2) For any value of i and k transform the column $(i, 1\dots N, k)$
- 3) For any value of i and j transform the column $(i, j, 1\dots N)$

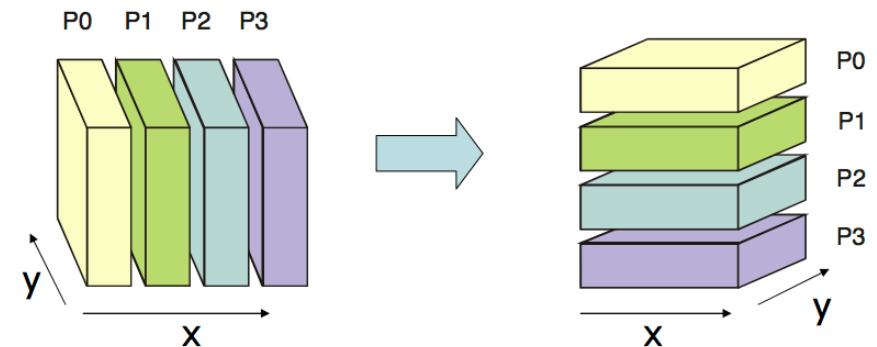
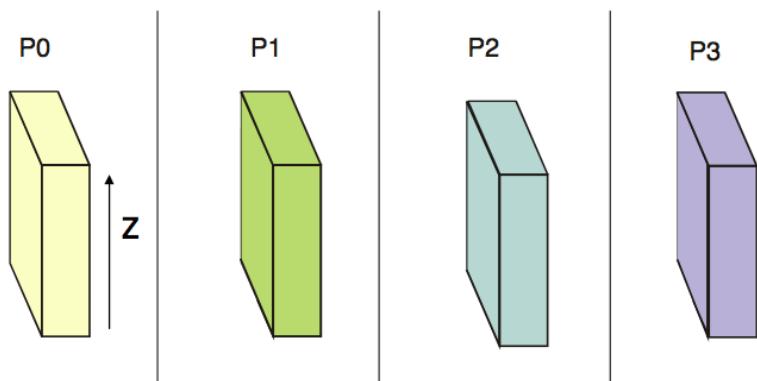
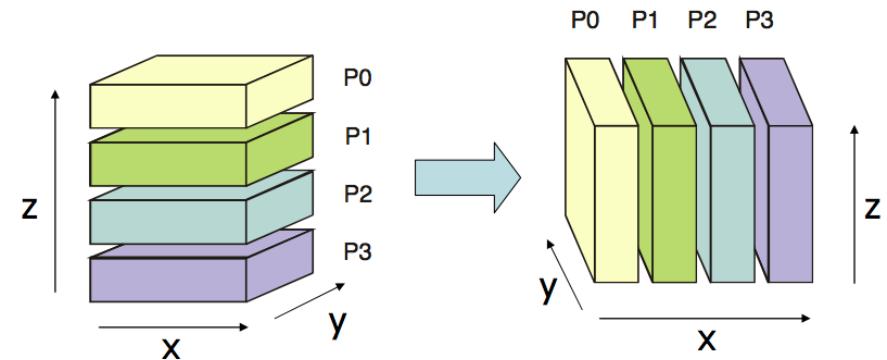
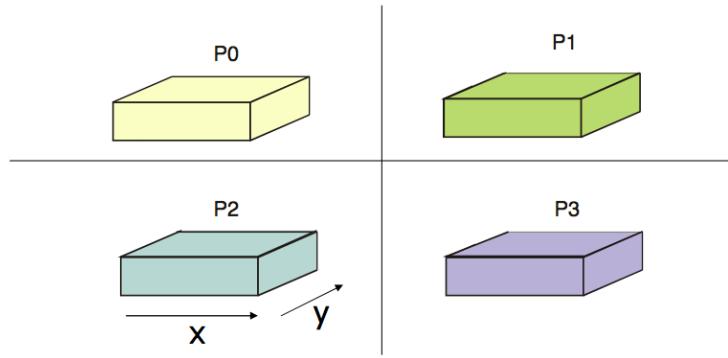
$$f(x, y, z) = \frac{1}{N_z N_y N_x} \sum_{z=0}^{N_z-1} \underbrace{\left(\sum_{y=0}^{N_y-1} \underbrace{\left(\sum_{x=0}^{N_x-1} F(u, v, w) e^{-2\pi i \frac{xu}{N_x}} \right) e^{-2\pi i \frac{yu}{N_y}} \right)}_{\text{DFT long } x\text{-dimension}} e^{-2\pi i \frac{zu}{N_z}}$$

$\underbrace{\phantom{\sum_{x=0}^{N_x-1}}}_{\text{DFT long } y\text{-dimension}}$
 $\underbrace{\phantom{\sum_{y=0}^{N_y-1}}}_{\text{DFT long } z\text{-dimension}}$

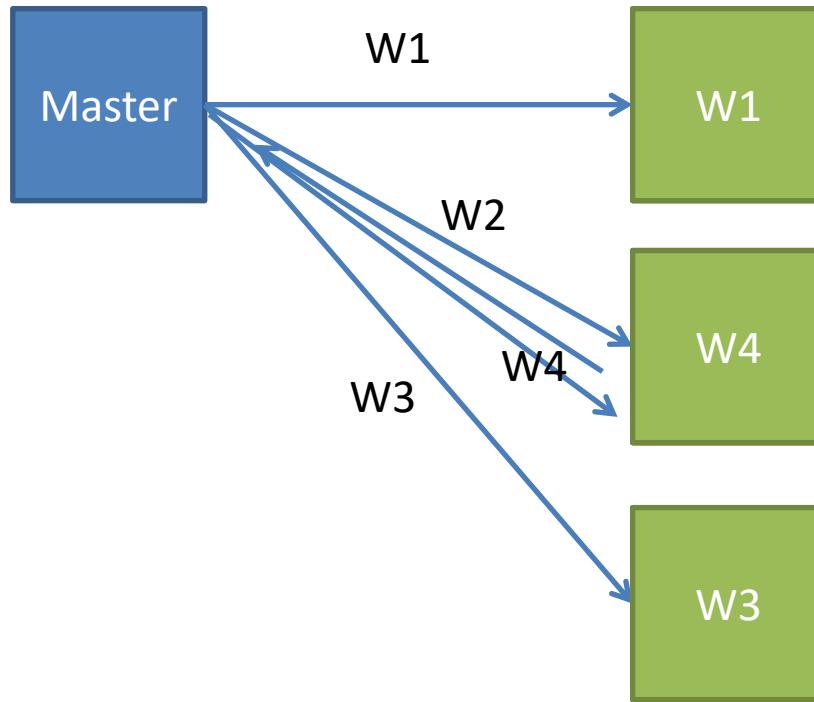
Parallel 3DFFT / 1



Parallel 3DFFT / 2



Master/Slave



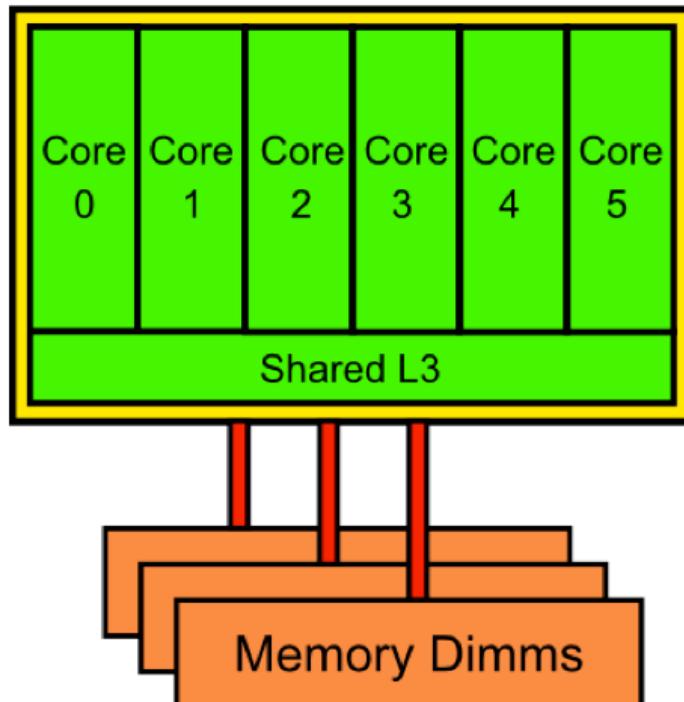


Single Program on Multiple Data

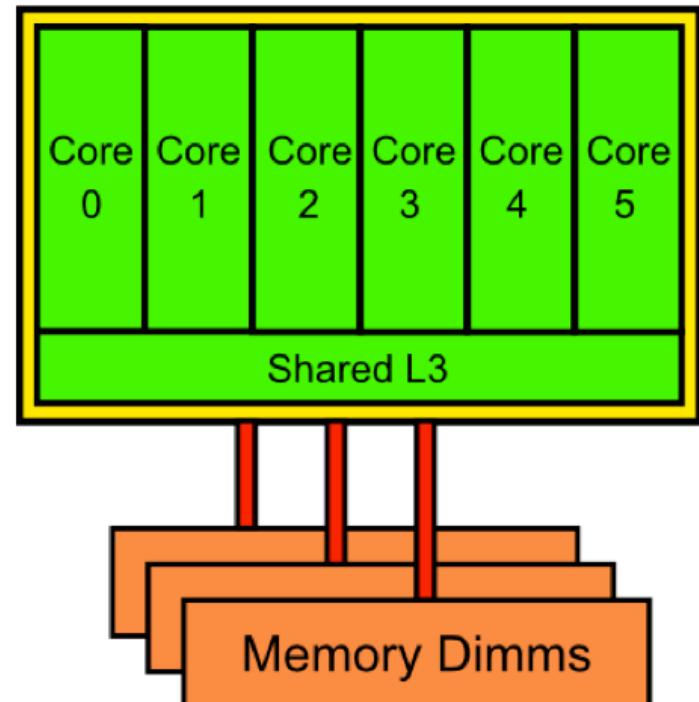
- performing the same program (set of instructions) among different data
- Same model adopted by the MPI library
 - “yum install openmpi”, “apt-get install openmpi”, etc...
- A parallel tool is needed to handle the different processes working in parallel
- The MPI library provides the *mpirun* application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

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

mynode01



mynode02





```
[igirotto@mynode01 ~]$ mpirun -np 12 /bin/hostname
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++ ){

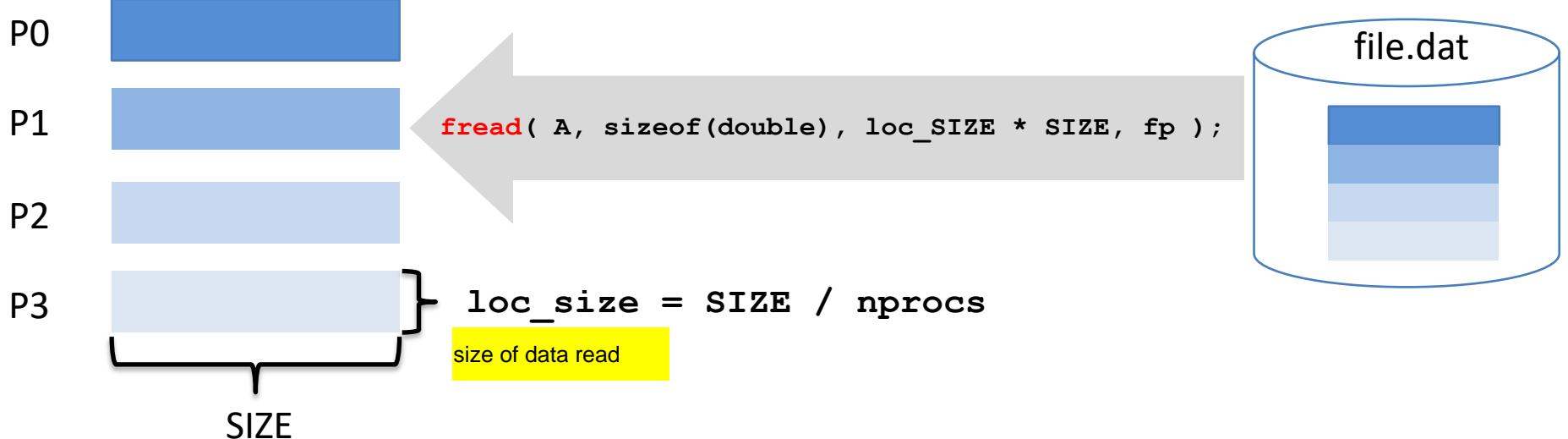
        A[i] = (double) ( rand() % 1000 + 1 );
    }

    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 write distributed data on a FILE?

P0



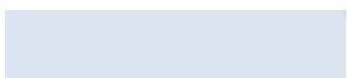
P1



P2

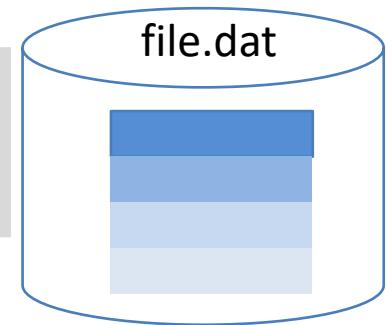


P3



SIZE

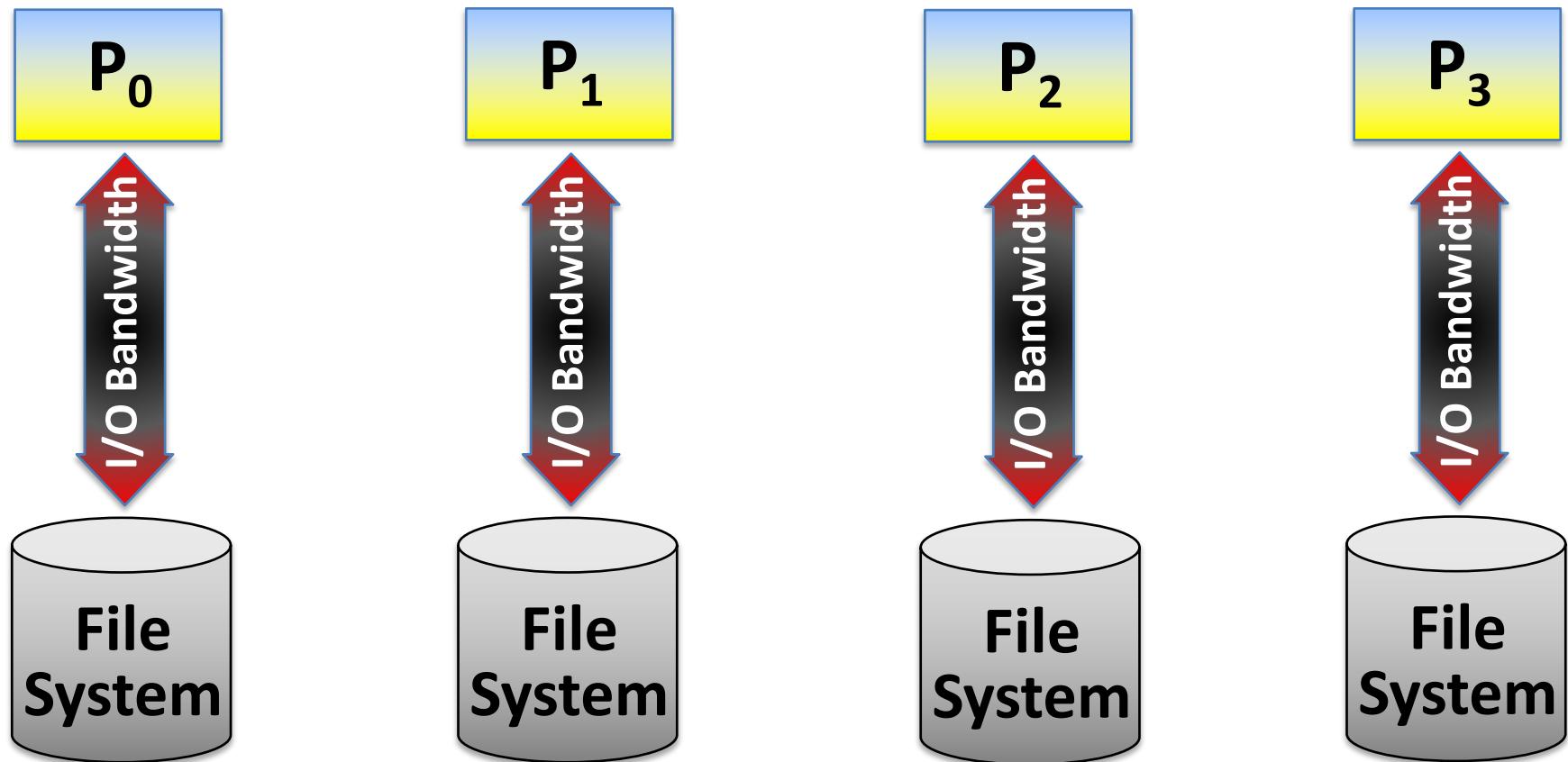
```
buf_size_bytes = loc_SIZE*SIZE*sizeof(double);  
  
fseek( fp, rank * buf_size_bytes, SEEK_SET );  
  
fread( A, sizeof(double), loc_SIZE * SIZE, fp );  
  
loc_size = SIZE / nprocs
```



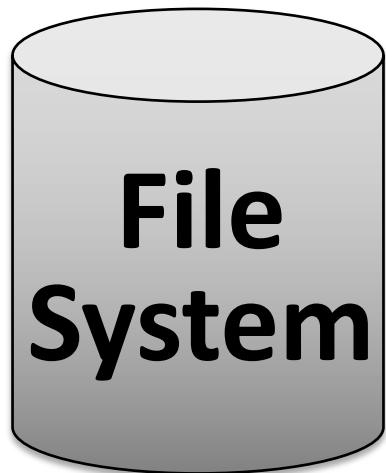
The `fseek()` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by `whence`. If `whence` is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start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

bottlenecks: I/O Bandwidth. At the end merging the files in serial ways at the end can take hundreds of ages

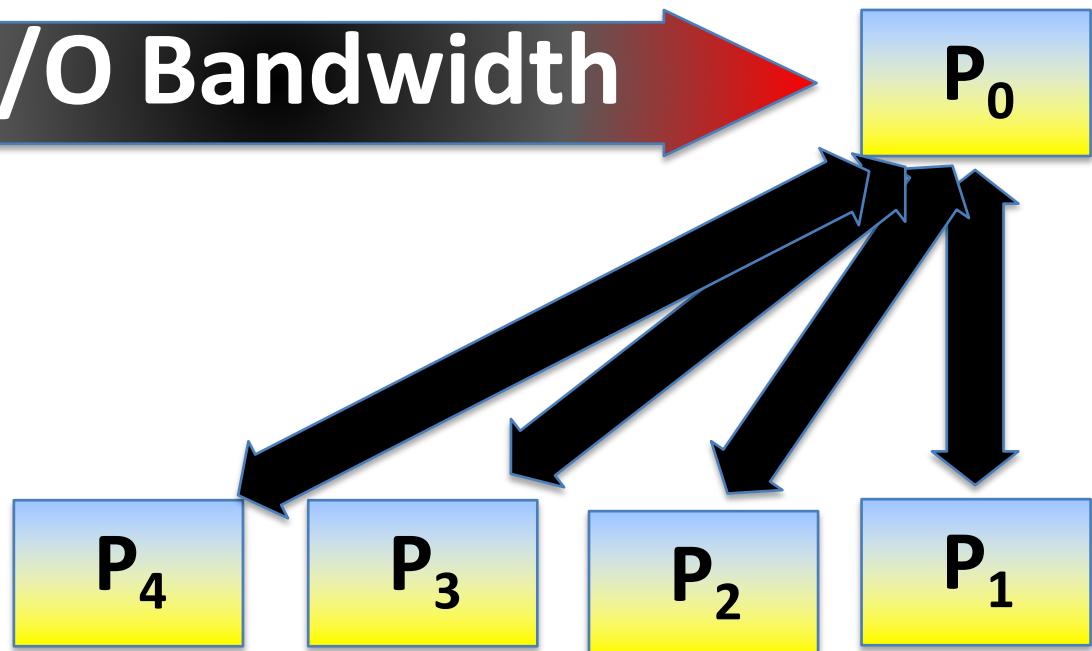
Parallel I/O



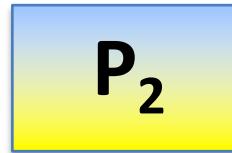
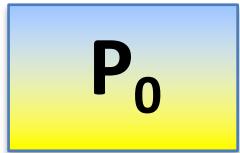
Parallel I/O



I/O Bandwidth

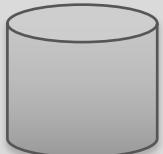


Parallel I/O



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 Sparse 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

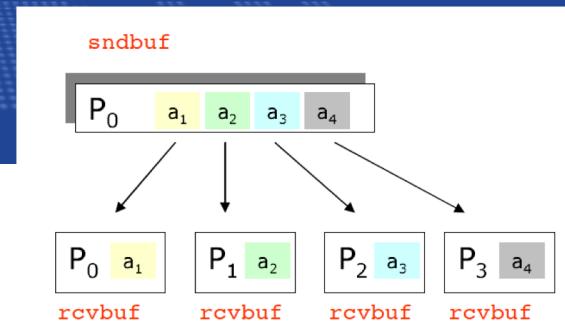


MPI_Scatter

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

don't use it: it doesn't scale with the size of procs because P0 needs to store all the data

```
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)
```



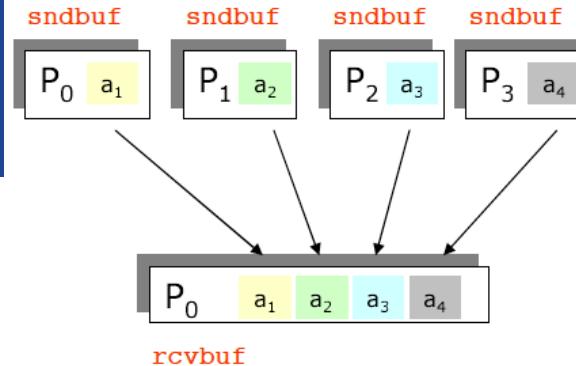
MPI_Gather

NOPE

NOPE

NOPE

NOPE



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

Blocking: the communication requires perfect synchro of the 2 procs, so it's likely one wait for the other, and so it is BLOCKED. Not doing computation

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

Opposite idea: 1 send the data and goes on executing instructions. It's dangerous, but can be profitable. Dangerous: you can modify data before the other pro

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

until wait u can go on without problems. wait assures synchro

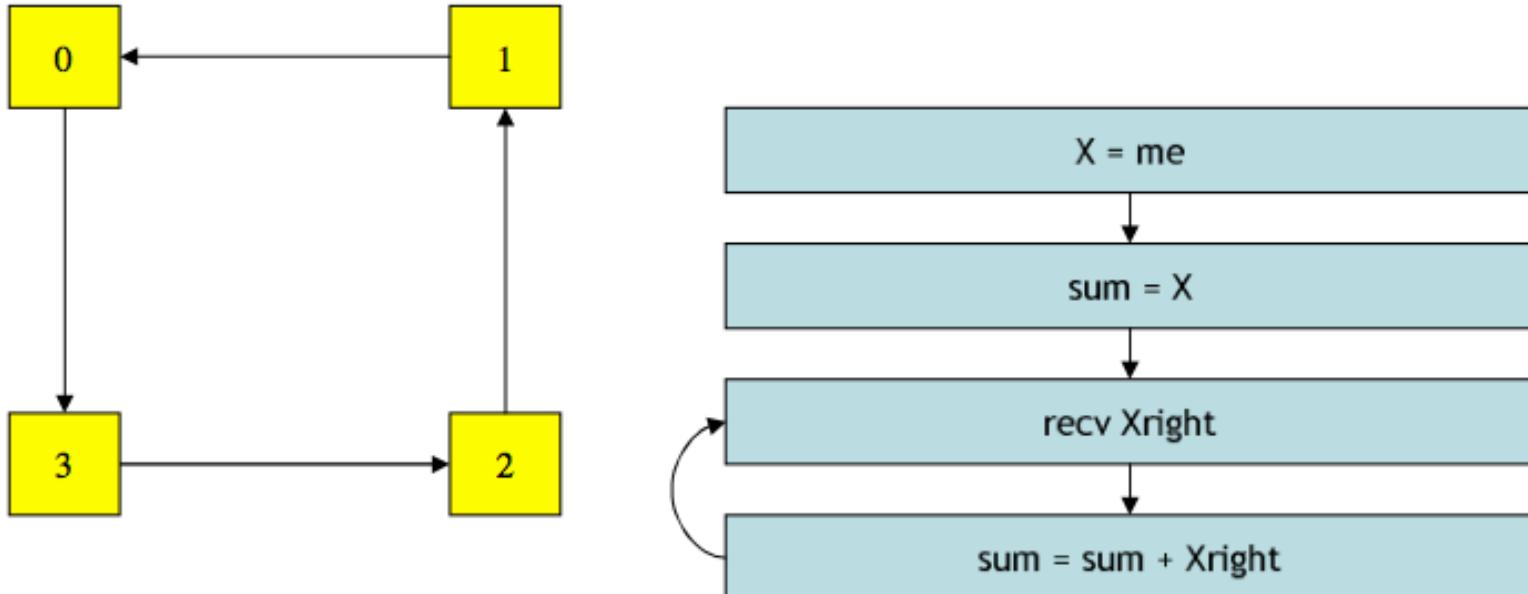
MPI_WAIT(request, status, ierr)

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

- 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).