Overview of Common Strategies for Parallelization

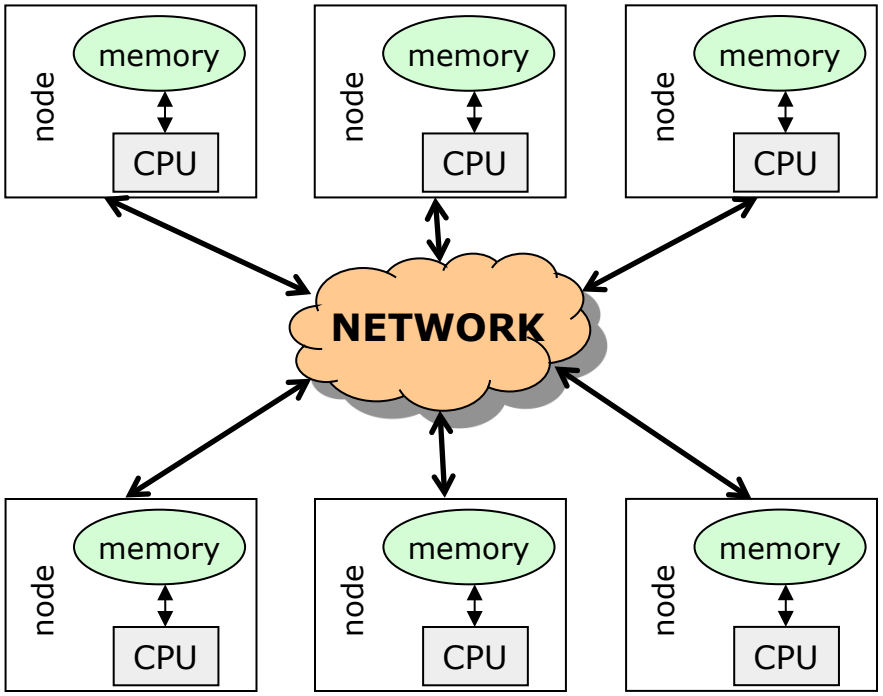
Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

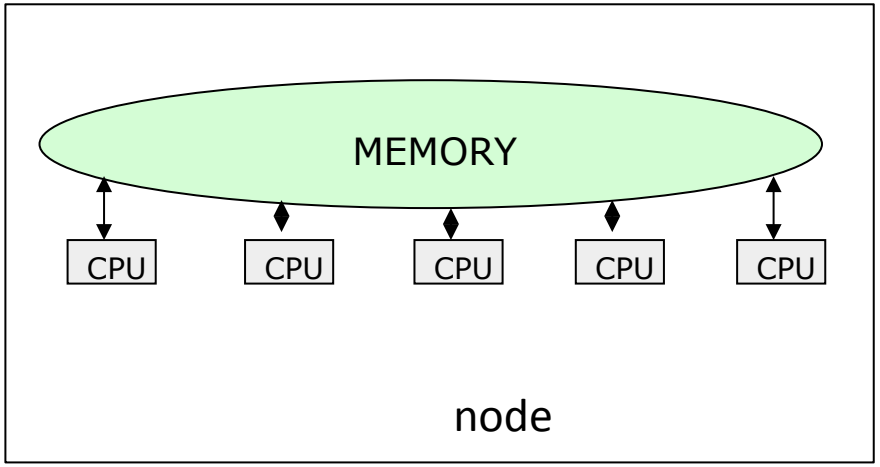
International Centre for Theoretical Physics (ICTP)

Parallel Architectures

- Distributed Memory



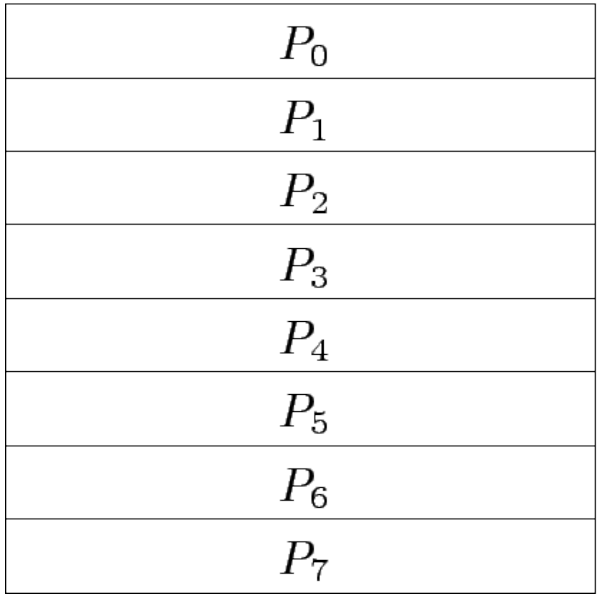
- Shared Memory



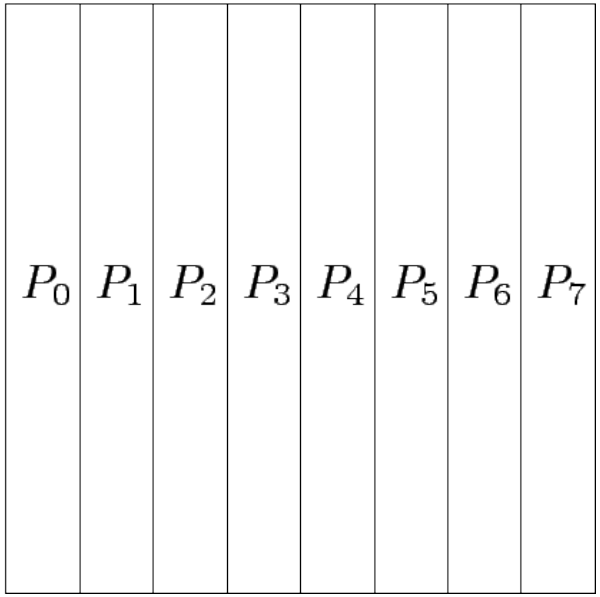
Static Data Partitioning

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

row-wise distribution



column-wise distribution

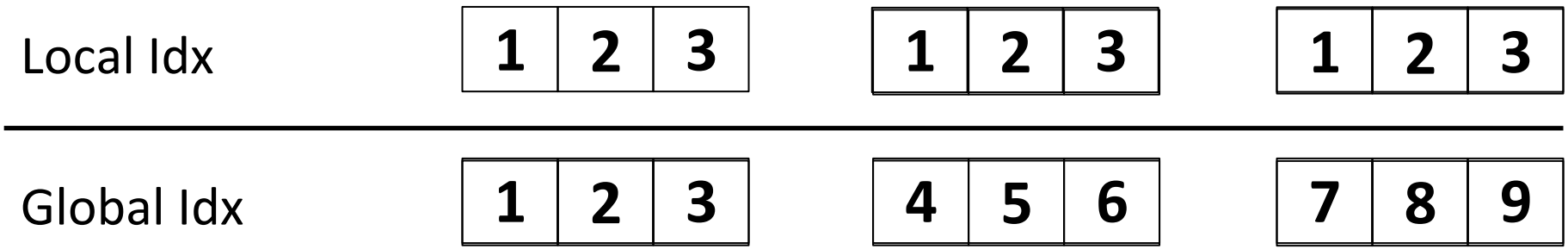


Distributed Data Vs Replicated Data

- 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

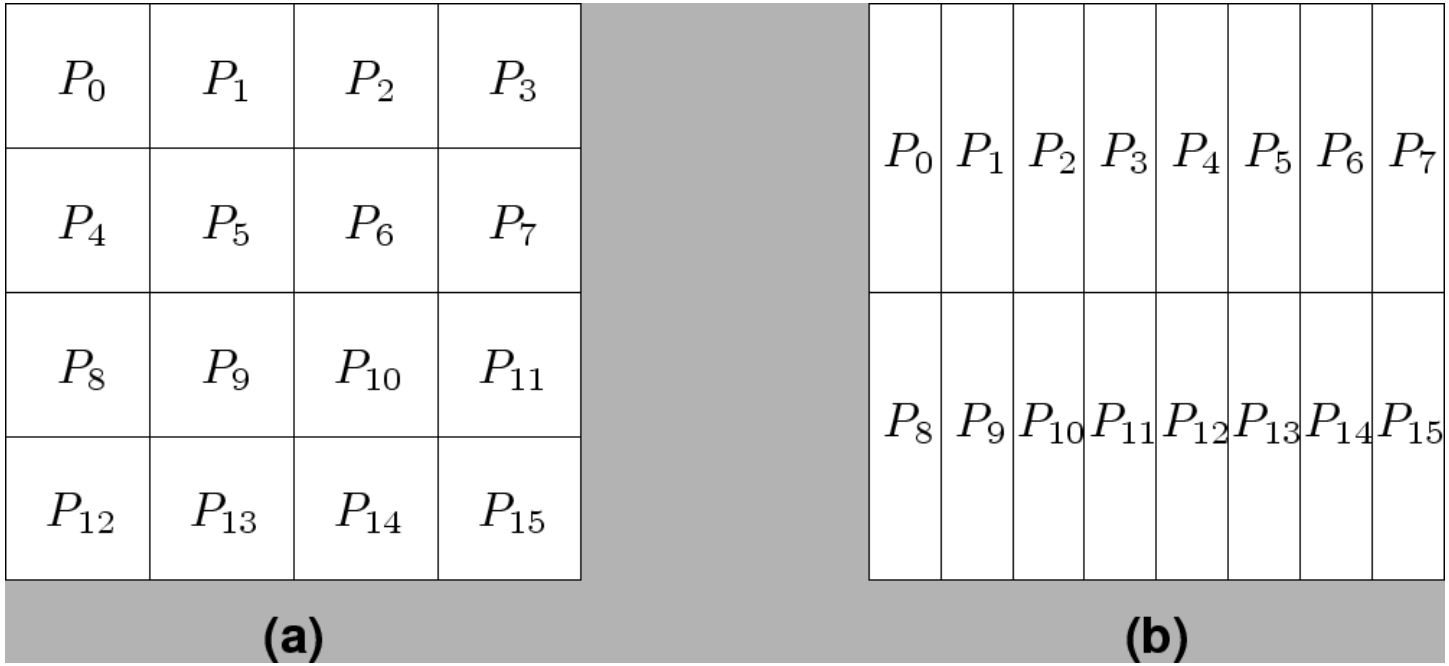
Global Vs Local Indexes

- 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)



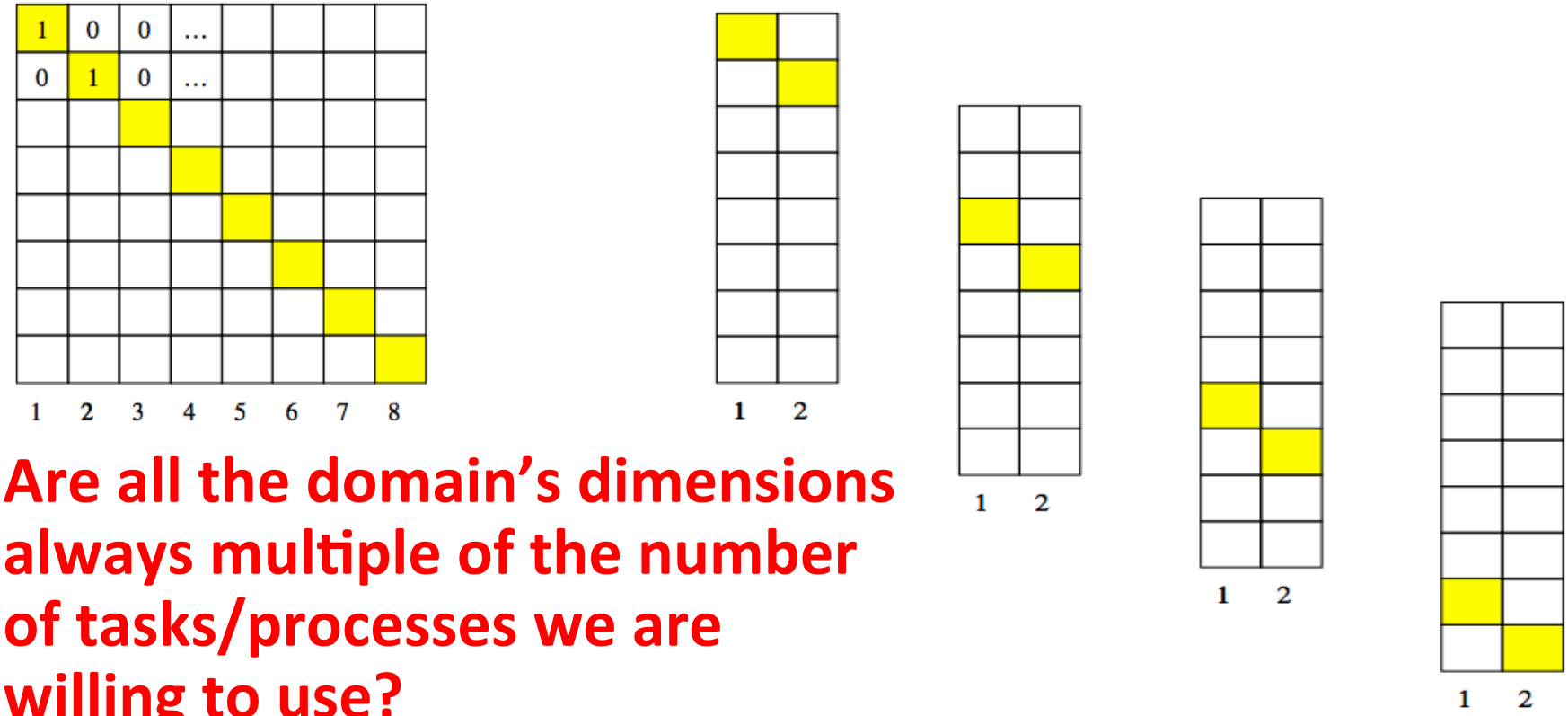
Block Array Distribution Schemes

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!!

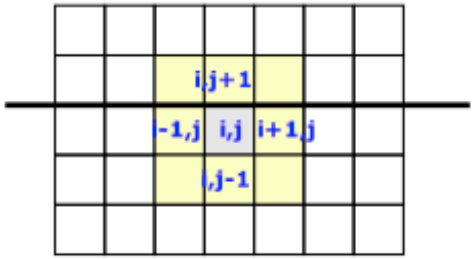
Collaterals to Domain Decomposition /1



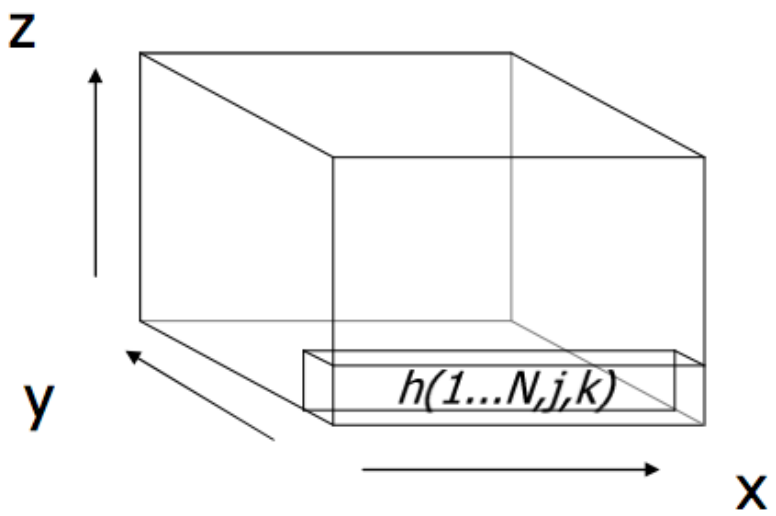
Are all the domain's dimensions always multiple of the number of tasks/processes we are willing to use?

Collaterals to Domain Decomposition /2

sub-domain boundaries



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

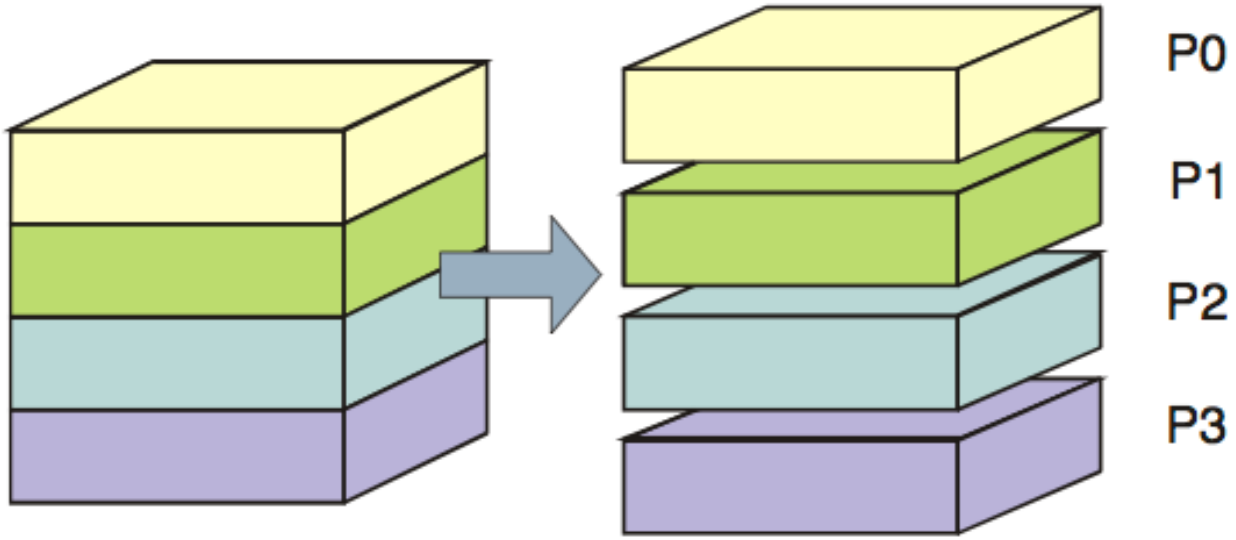
DFT long z-dimension

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

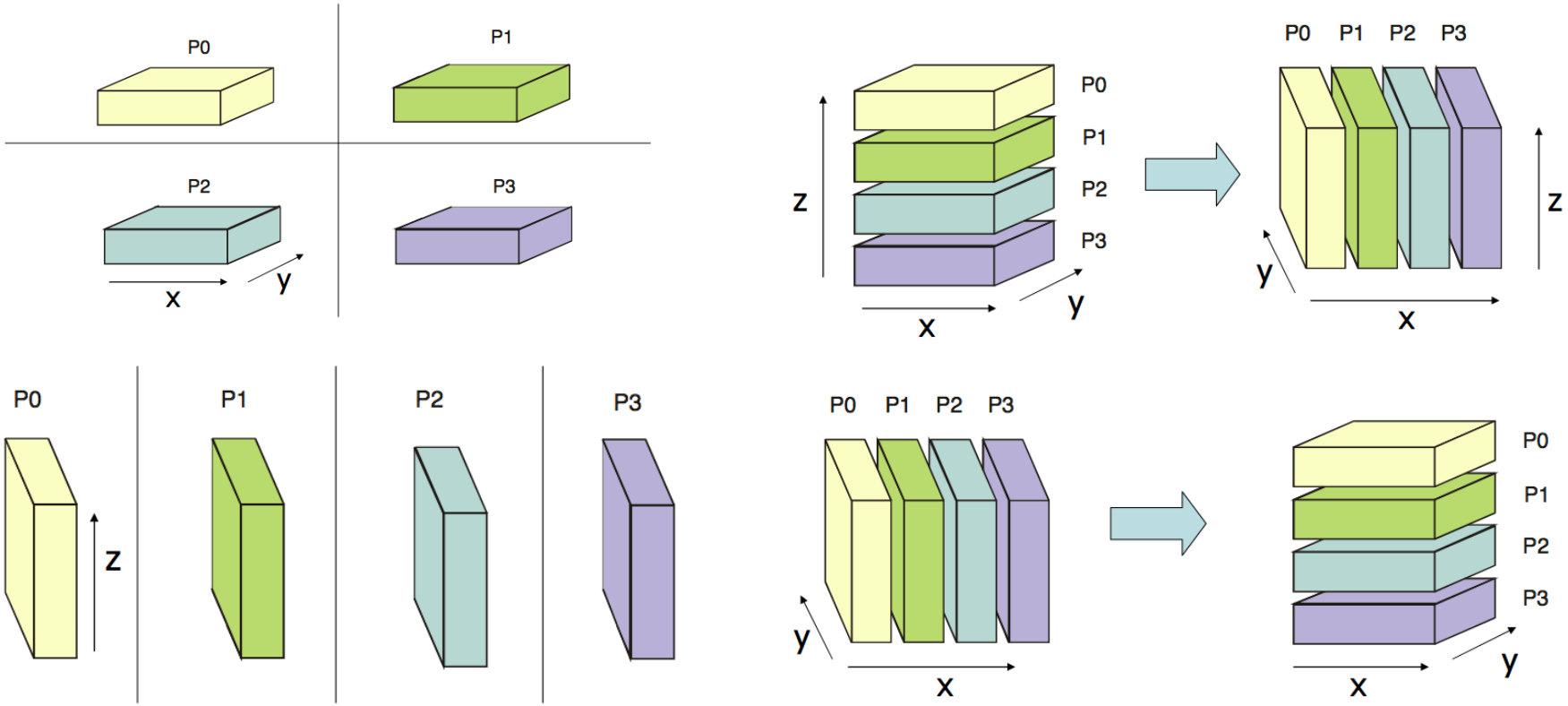
2) For any value of ***i*** and ***k*** transform the column (i, 1…N, k)

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

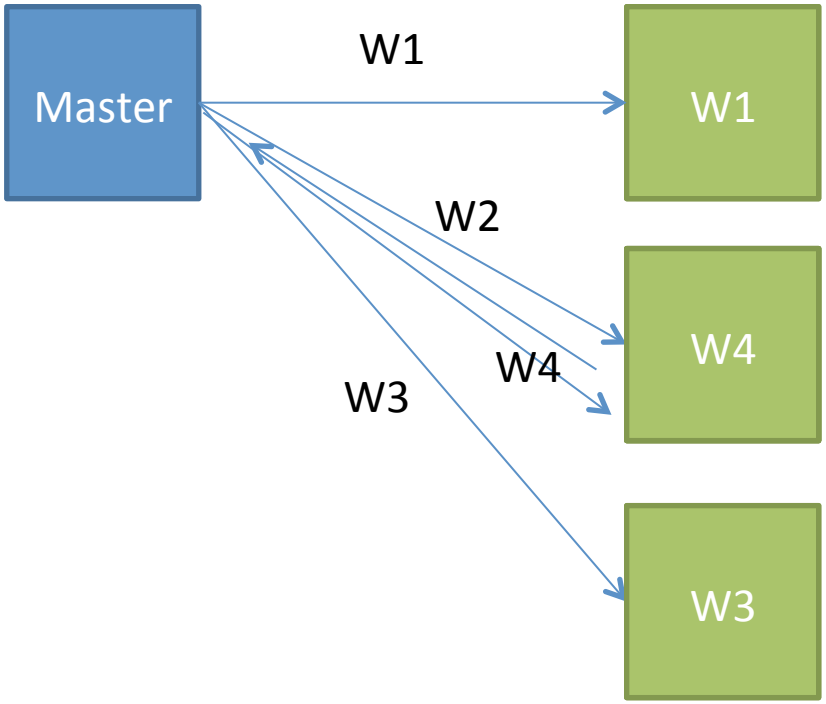
Parallel 3DFFT / 1



Parallel 3DFFT / 2



Master/Slave

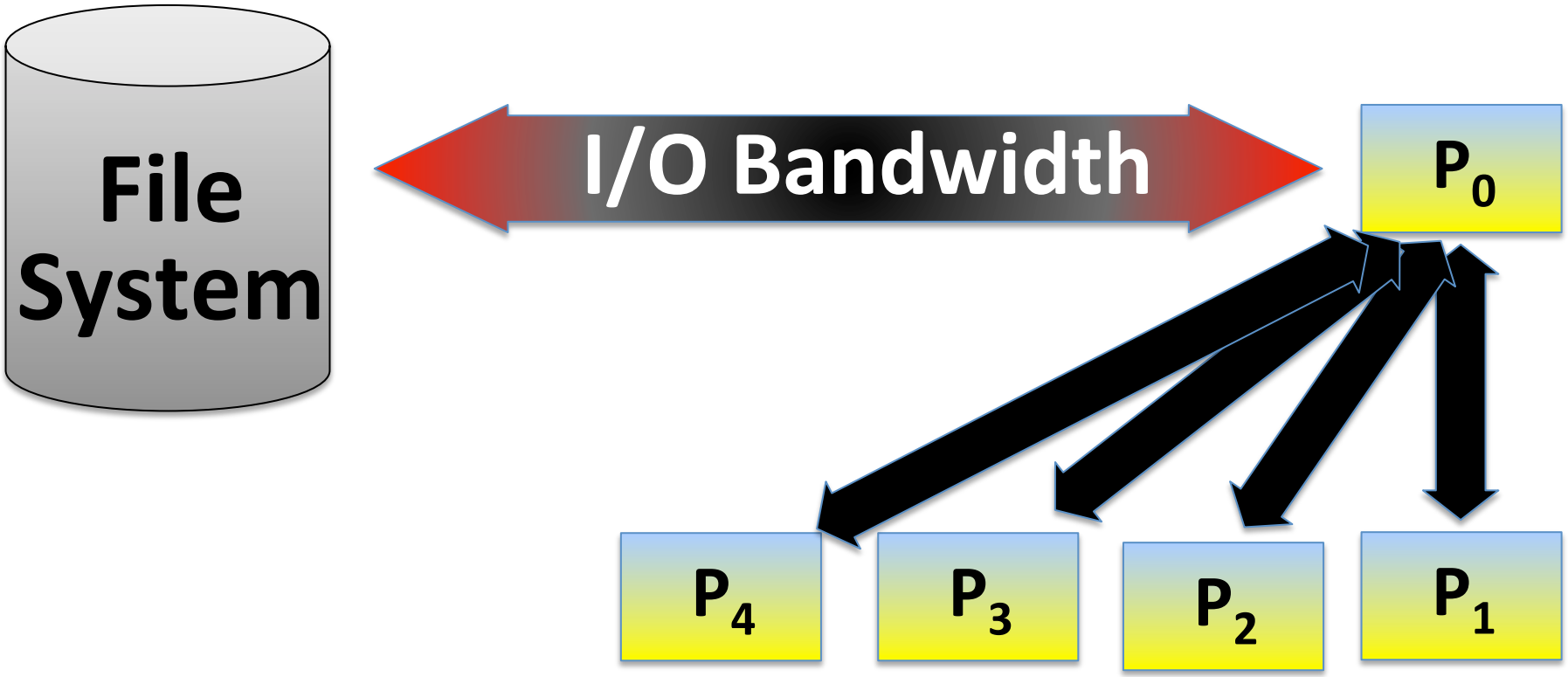


Task Farming

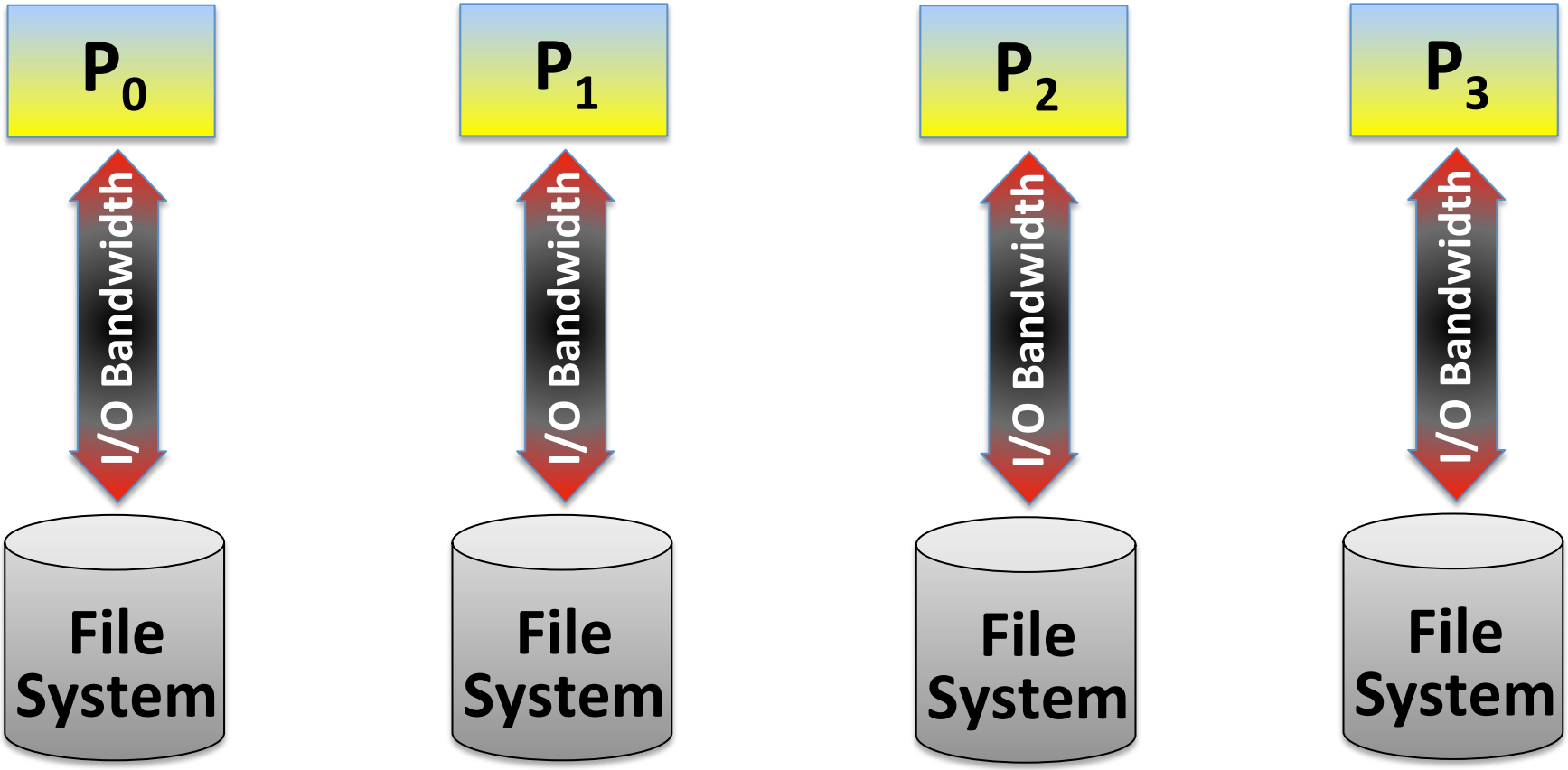
- Many independent programs (tasks) running at once
  - each task can be serial or parallel
  - "independent" means they don't communicate directly
  - Processes possibly driven by the mpirun framework

```
[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
```

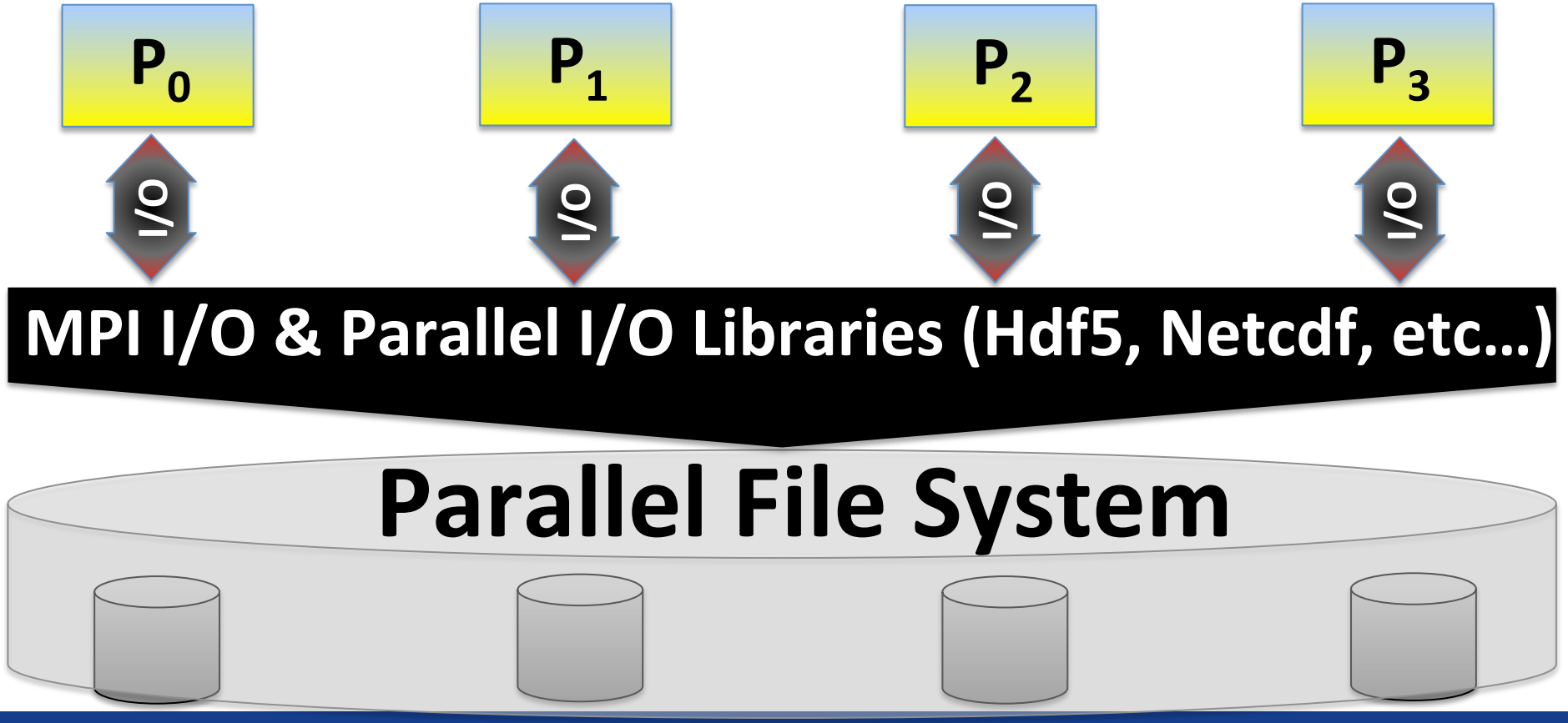
Parallel I/O



Parallel I/O

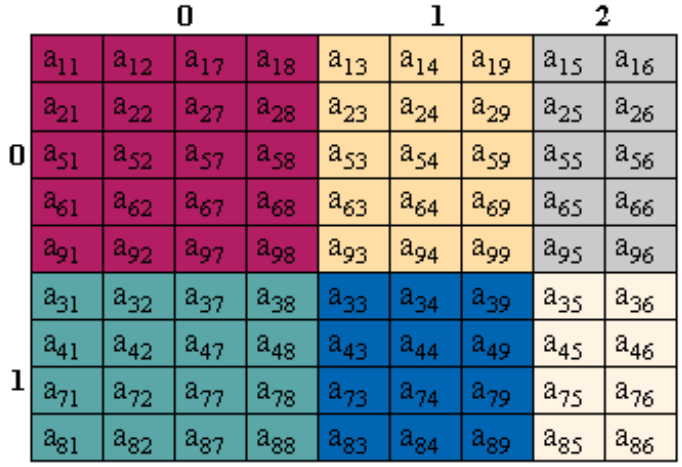

P0
P1
P2
P3
I/O Bandwidth
I/O Bandwidth
I/O Bandwidth
I/O Bandwidth
File System
File System
File System
File System

Parallel I/O


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

block cyclic distribution

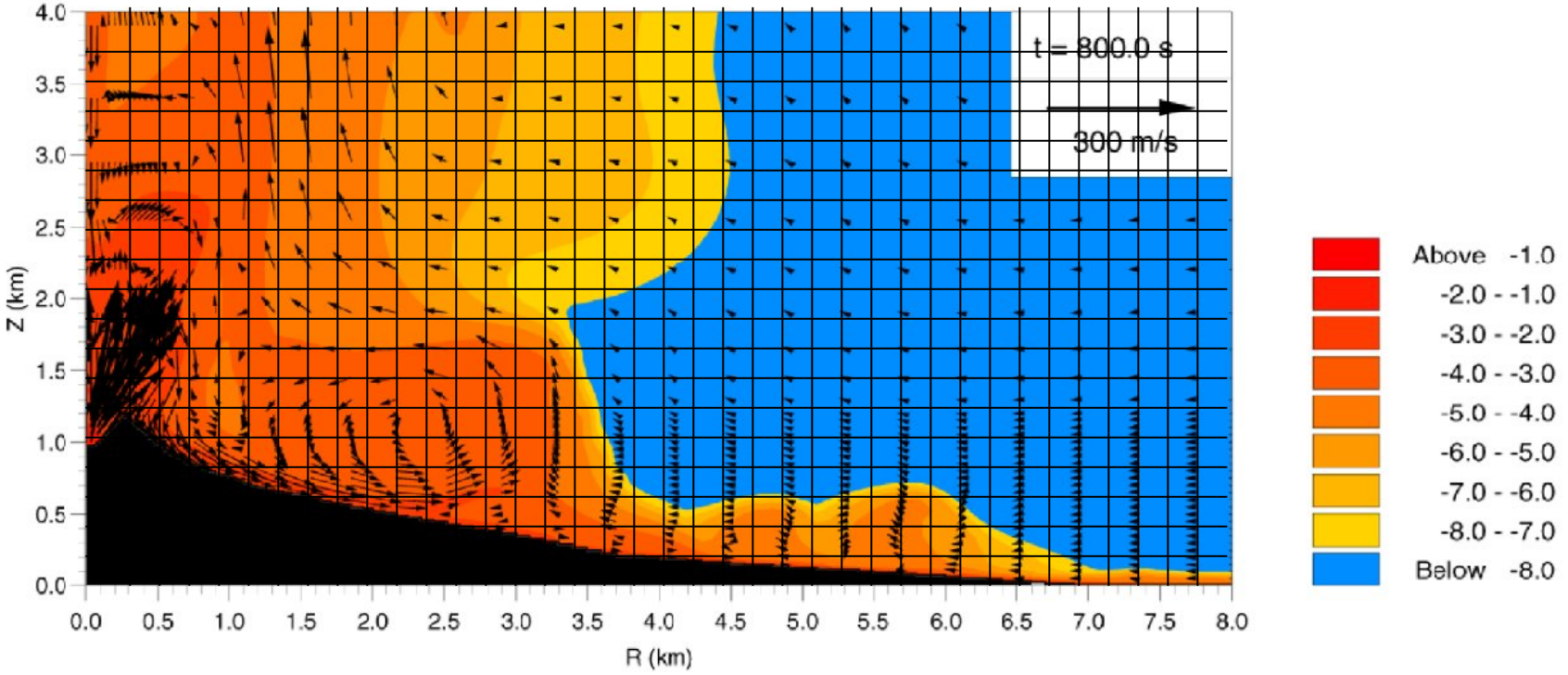
Global View



Local (distributed) View

Parallelization Strategies

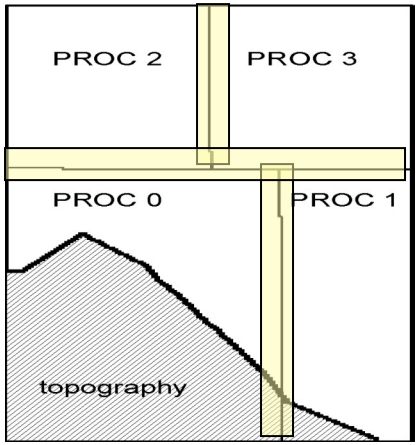
In a simulation the space and time are discretized, and the transport equations are solved numerically on a discrete grid



Computational Grid (Nx * Nz), topography, gas and particle fields

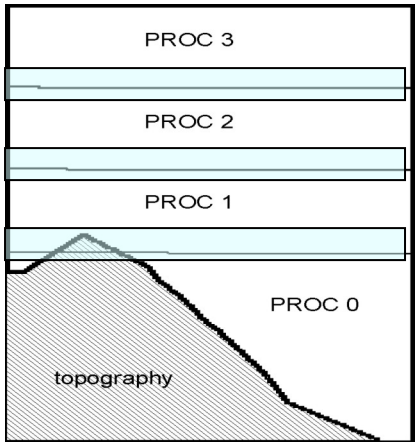
Parallelization Strategies

The main strategy is to define sub grids and distribute them to the available processors. Intermediate results on local data need to be exchanged across domain boundaries the solve the equations defined in the global domain.



Block like distribution

less overall communication



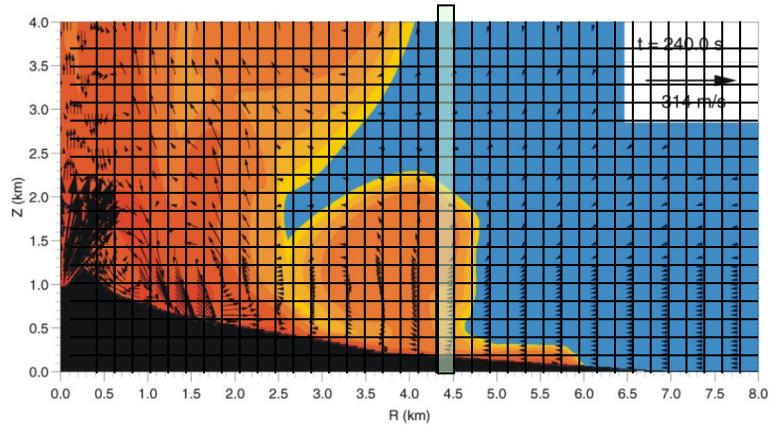
layer like distribution,

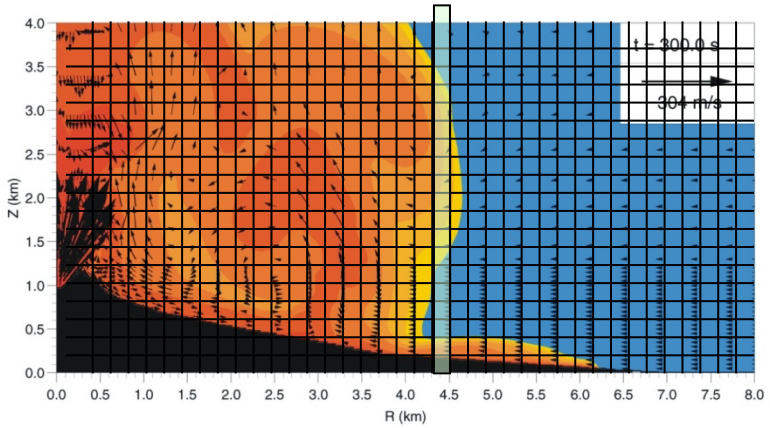
comunication only with nearest neighbours

Time evolution

$s(t+\Delta t) = F(s(t), s(t-\Delta t), ..)$

System status at time $t+\Delta t$ is a Function of the System at previous time values t, $t-\Delta t$, ecc…
Usually we need only information on system from t.





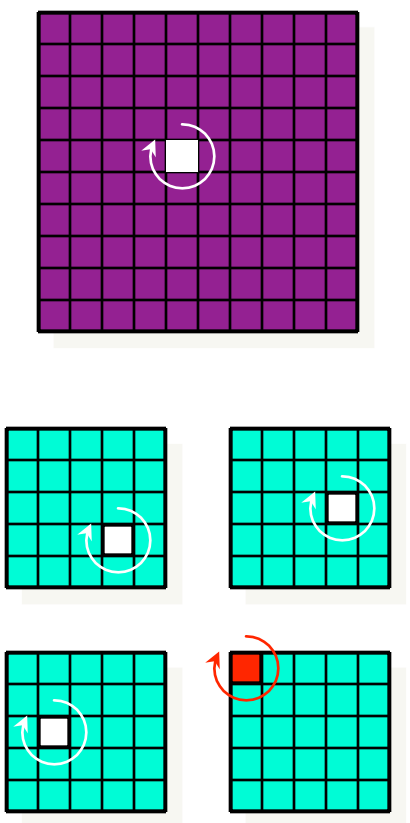
Ghost Cells

- In many algorithms used to solve partial differential equations, the vaule of a given field in a given point/ cell is updated using the values of the neighbouring cells.

- Distributing the field to different PE implies that some point lays close to a boundary between two task.

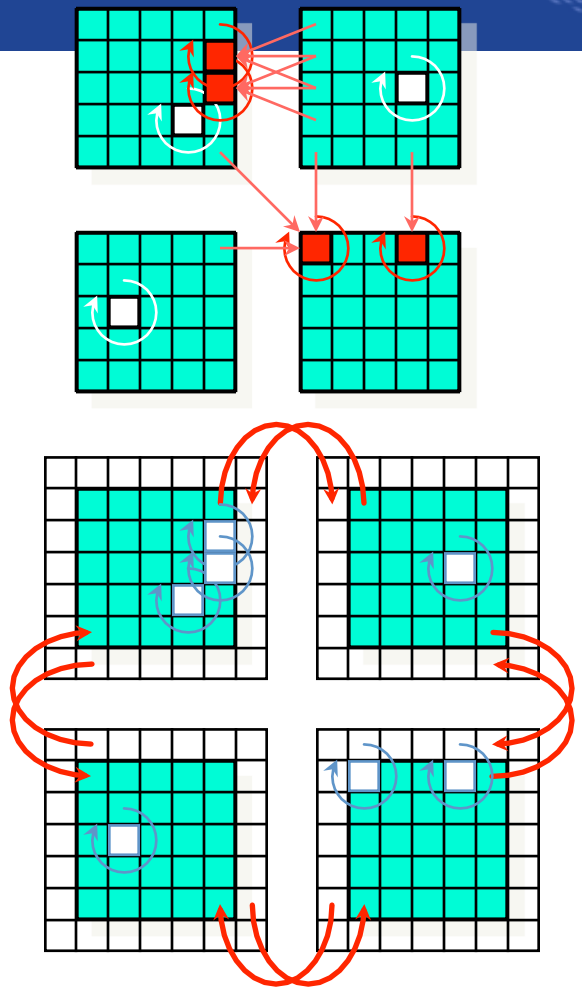
- The boundary elements require values stored in the memory of another task

- At first approach a developer could be tended to perform a communication for every element!!! => No!!!



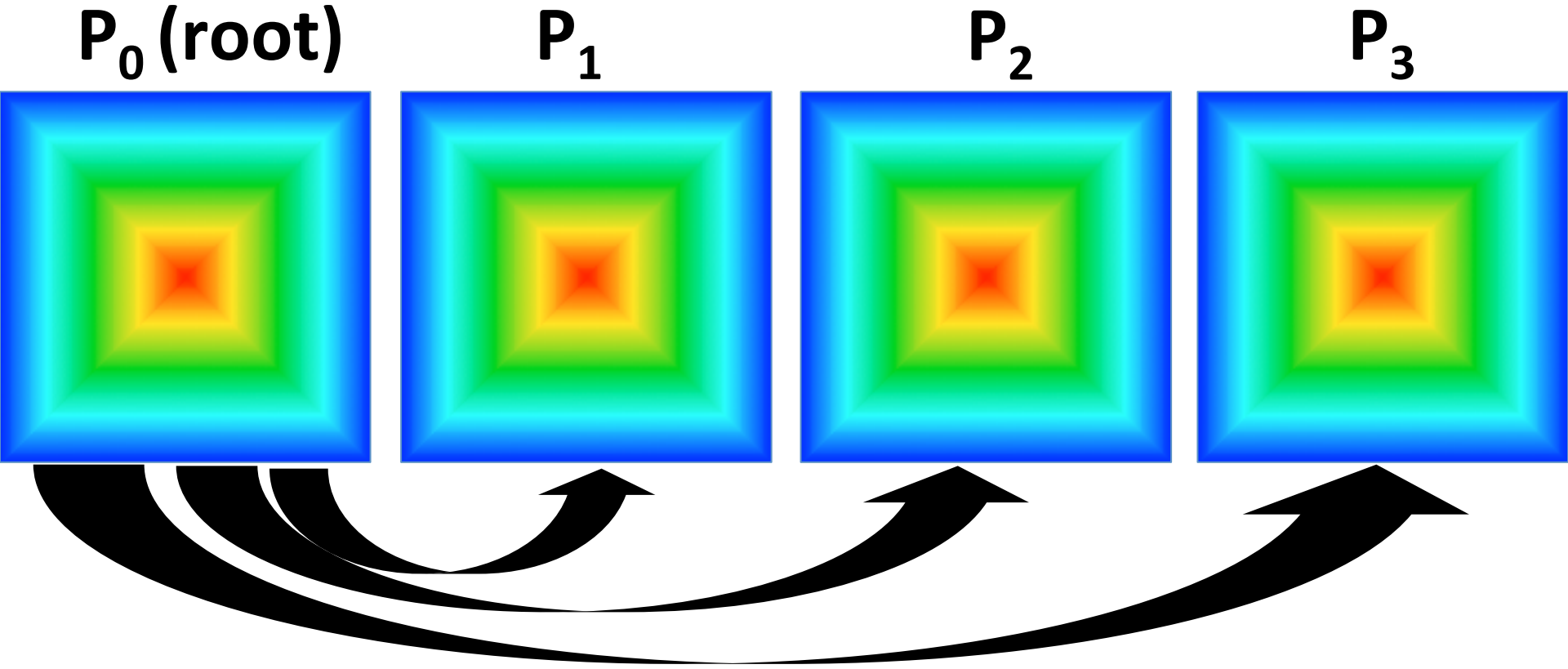
Ghost Cells /2

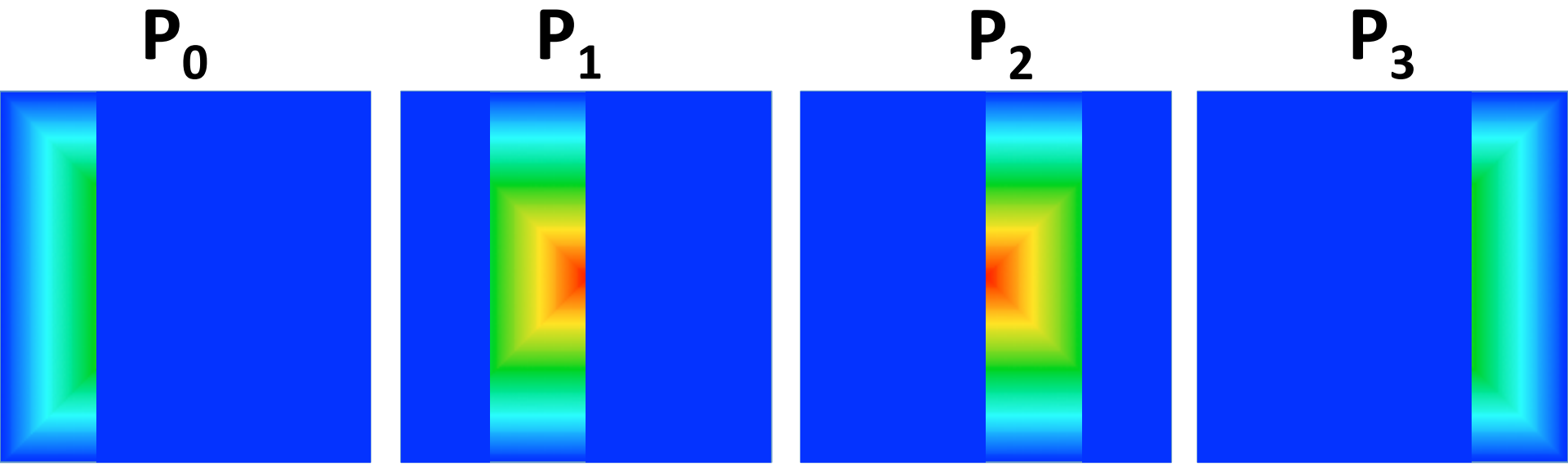
- To solve the problem, each PE extend its domain to contain a copy of the value of its neighbouring PEs
- The “ghost cells” are updated with a single communication before the updating cycle
- The locality of the computation is restored.



P_0

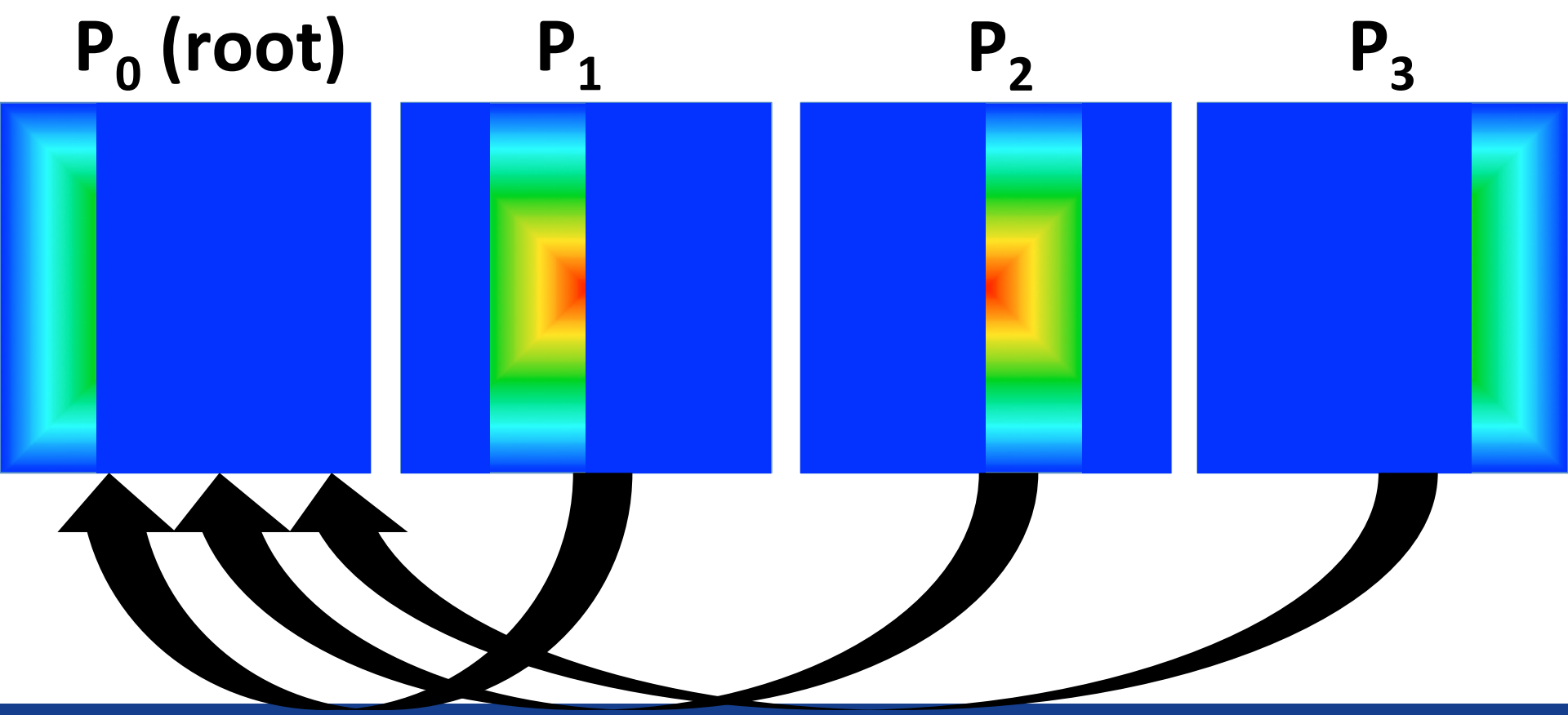
call MPI_BCAST(...)





call evolve(dtfact)

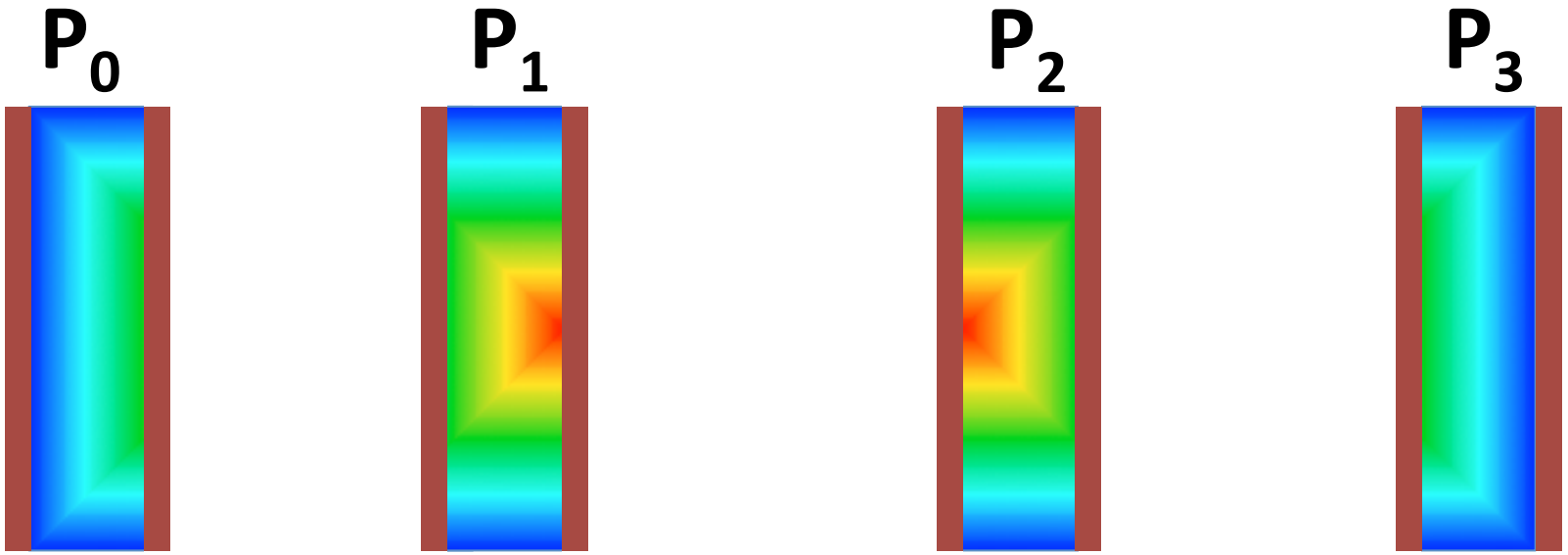
call MPI_REDUCE(..., MPI_SUM, ...)



The Transport Code - Parallel Version

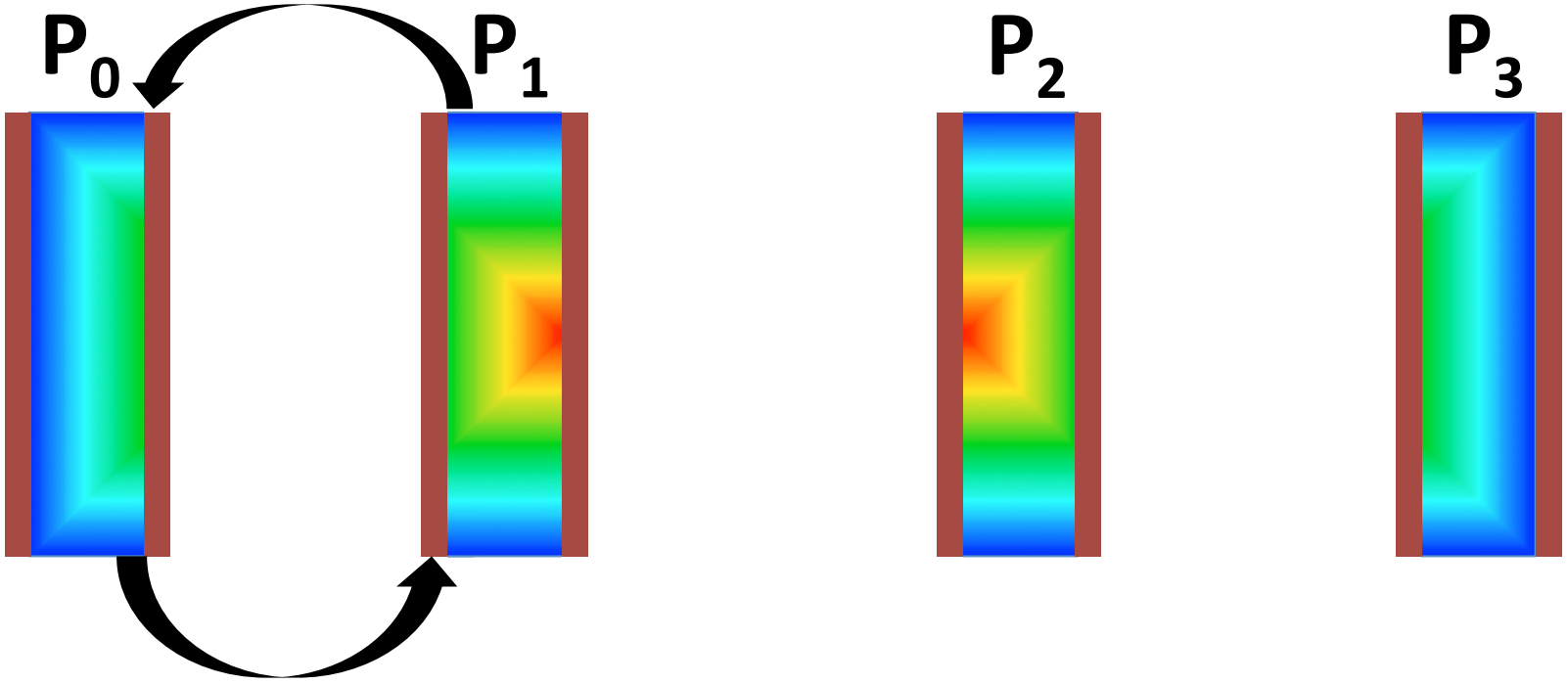
- Replicated data
- Compute domain (and workload) distribution among processes
- Master-slaves: P_0 drives all processes
- Large amount of data communication
  - at each step P_0 distribute data to all processes and collect the contribution of each process
- Problem size scaling limited in memory capacity

The Transport Code - Parallel Version



call evolve(dtfact)

Data exchange among processes



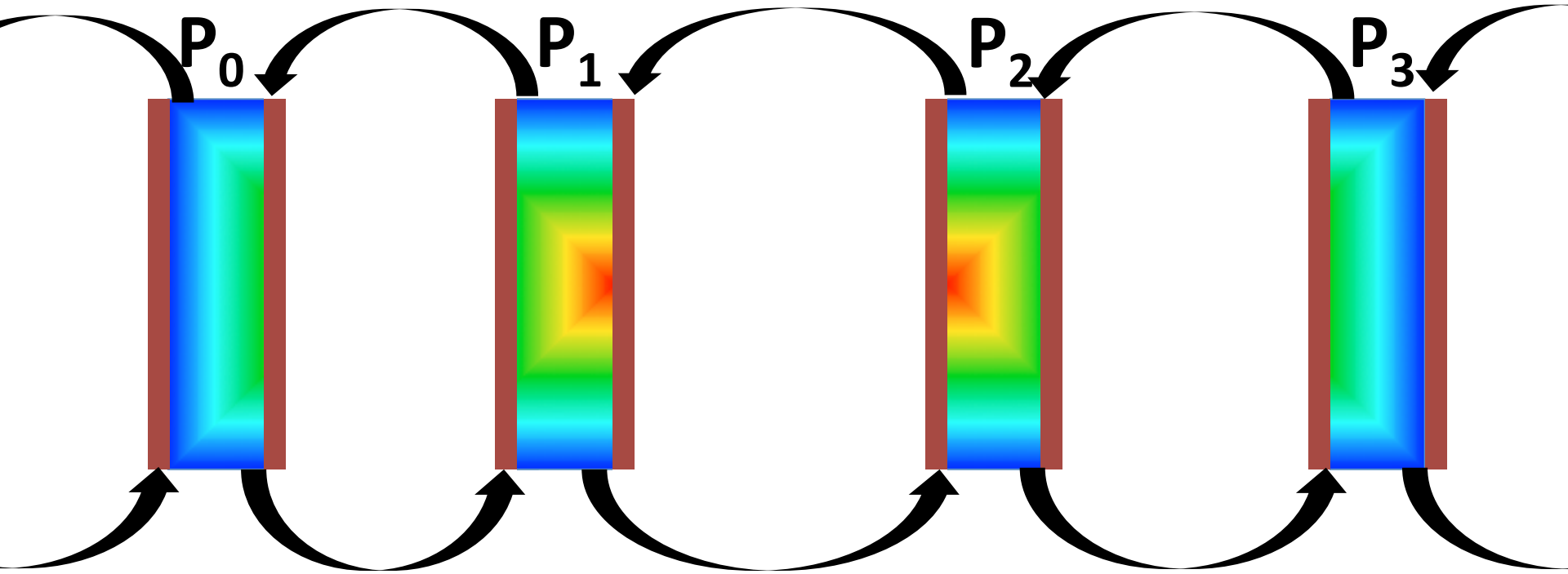
```
PROGRAM send_recv
      INCLUDE ‘mpif.h‘
      INTEGER :: ierr, myid, nproc, status(MPI_STATUS_SIZE)
      REAL A(2)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      IF( myid .EQ. 0 ) THEN
            A(1) = 3.0
            A(2) = 5.0
            CALL MPI_SEND(A, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            CALL MPI_RECV(A, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
            WRITE(6,*) myid,’: a(1)=’,a(1),’ a(2)=’,a(2)
      END IF

      CALL MPI_FINALIZE(ierr)
END
```

```
PROGRAM error_lock
      INCLUDE ‘mpif.h‘
      INTEGER :: ierr, myid, nproc, status(MPI_STATUS_SIZE)
      REAL :: A(2), B(2)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      IF( myid .EQ. 0 ) THEN
            a(1) = 2.0
            a(2) = 4.0
            CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            a(1)  = 3.0
            a(2)  = 5.0
            CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
      END IF
      WRITE(6,*) myid, ’: a(1)=’, a(1), ’ a(2)=’, a(2)
      CALL MPI_FINALIZE(ierr)
END
```

```
PROGRAM error_lock
      INCLUDE ‘mpif.h‘
      INTEGER :: ierr, myid, nproc, status(MPI_STATUS_SIZE)
      REAL :: A(2), B(2)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      IF( myid .EQ. 0 ) THEN
            a(1) = 2.0
            a(2) = 4.0
            CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            a(1)  = 3.0
            a(2)  = 5.0
            CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
      END IF
      WRITE(6,*) myid, ’: a(1)=’, a(1), ’ a(2)=’, a(2)
      CALL MPI_FINALIZE(ierr)
END
```

```
PROGRAM error_lock
      INCLUDE ‘mpif.h‘
      INTEGER :: ierr, myid, nproc, status(MPI_STATUS_SIZE)
      REAL :: A(2), B(2)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      IF( myid .EQ. 0 ) THEN
            a(1) = 2.0
            a(2) = 4.0
            CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
            CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            a(1)  = 3.0
            a(2)  = 5.0
            CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
      END IF
      WRITE(6,*) myid, ’: a(1)=’, a(1), ’ a(2)=’, a(2)
      CALL MPI_FINALIZE(ierr)
END
```

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



proc_up = mod(proc_me + 1 , nprocs)

STANDARD NO-BLOCKING SEND - RECV

- Basic point-2-point communication routines in MPI.

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

MPI_IRECV(buf, count, type, source, tag, comm, req, ierr)

Buf array of MPI type **type**.

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

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

Dest (INTEGER) rank of the destination process / **Source** (INTEGER) rank of the source process

Tag (INTEGER) number identifying the message

Comm (INTEGER) communicator of the sender and receiver

Req (INTEGER) output, identifier of the communications handle

Ierr (INTEGER) error code

STANDARD NO-BLOCKING WAIT

- A call to this subroutine cause the code to wait until the communication pointed by req is complete
- Handler for no-blocking communication

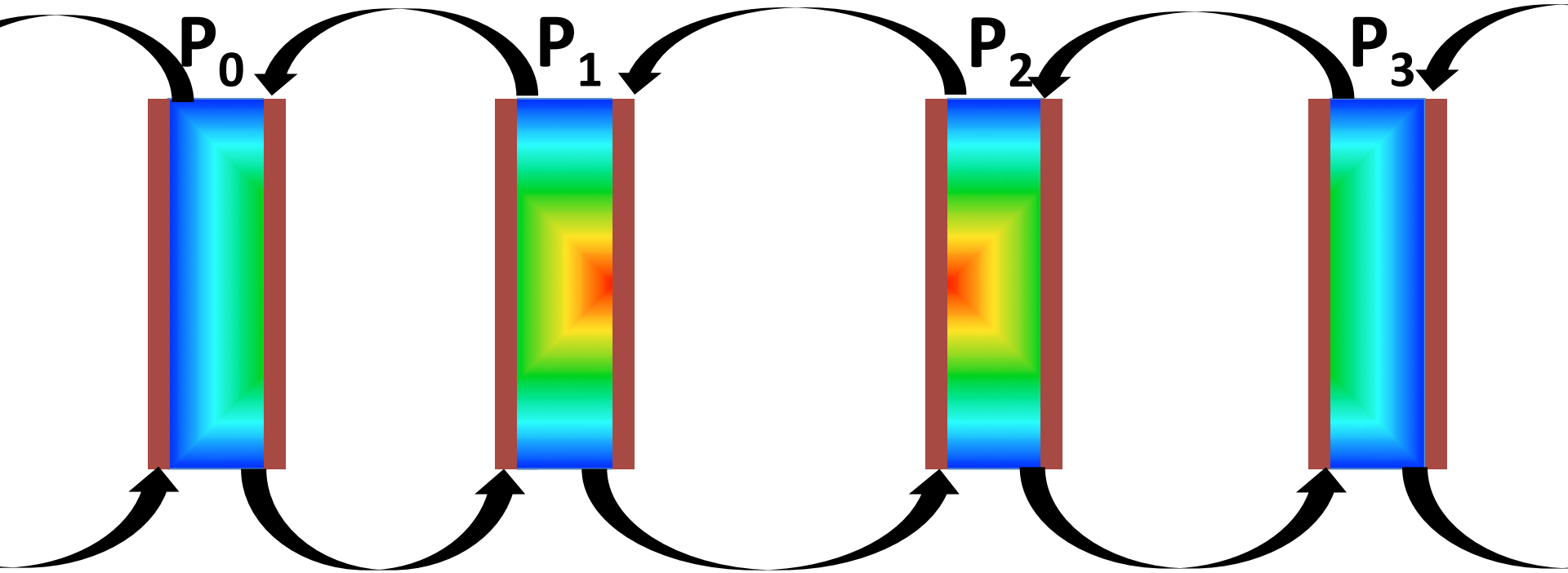
MPI_WAIT(req, status, ierr)

Req (INTEGER) output, identifier of the communications handle

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

Ierr (INTEGER) error code

The Transport Code - Parallel Version



The Transport Code - Parallel Version

- Distributed Data
- Global and Local Indexes
- Ghost Cells Exchange Between Processes
  - Compute Neighbor Processes
- Serialized Output on Process 0 (provided)

SendRecv

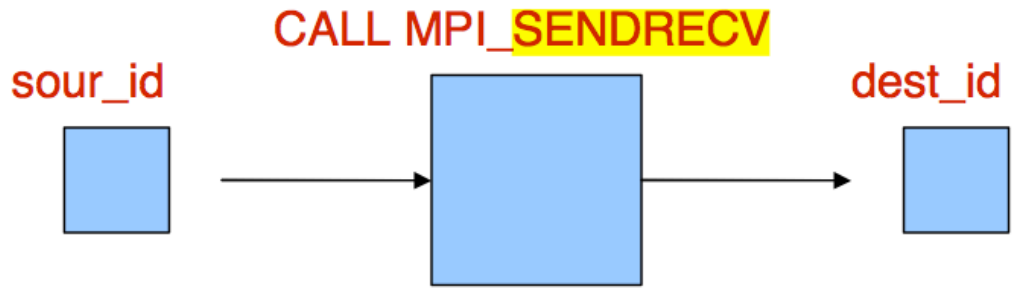
The easiest way to send and receive data without warring about deadlocks

Sender side

Fortran:

```
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type, dest_id, tag,
rcvbuf, rcv_size, rcv_type, sour_id, tag, comm, status, ierr)
```

Receiver side

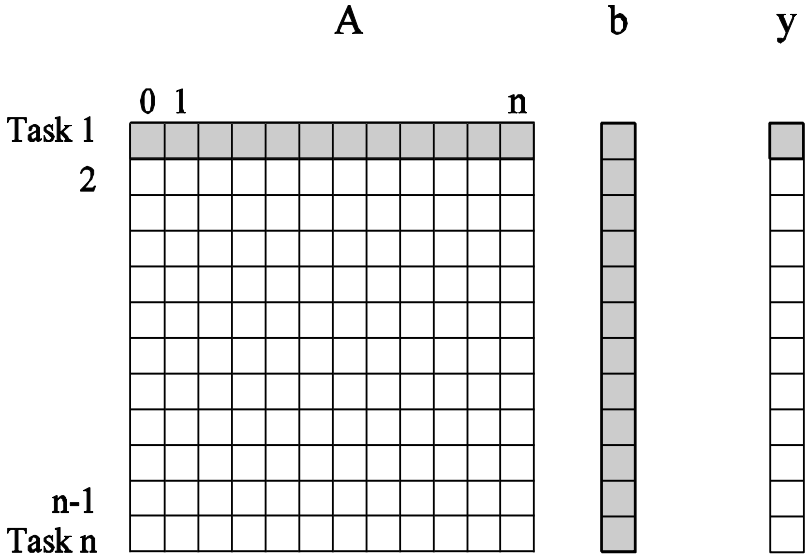


```
PROGRAM send_recv
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_SENDRECV(a, 2, MPI_REAL, 1, 10, b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SENDRECV(a, 2, MPI_REAL, 0, 11, b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
```

Communication Cycle

```
! right to left !
call MPI_SENDRECV(snd_buffer, ibuf, MPI_REAL, right, 1, &
     rcv_buffer, ibuf, MPI_REAL, left , 1, &
     comm_cart, istatus, ierr)

! left to right !
call MPI_SENDRECV (snd_buffer, ibuf, MPI_REAL, left, 1, &
     rcv_buffer, ibuf, MPI_REAL, right , 1, &
     comm_cart, istatus, ierr)
```

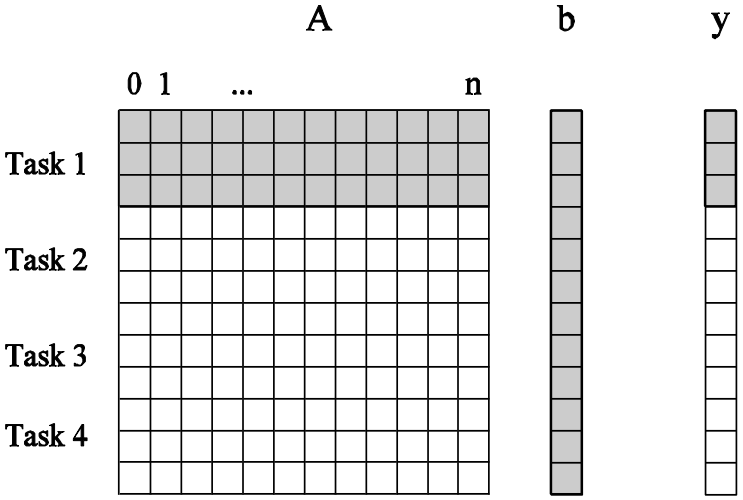
Replication vs Distribution



Under the hypothesis that the vector b is replicated, computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks.

Granularity of Task Decompositions

- The number of tasks into which a problem is decomposed determines its granularity.
- Decomposition into a large number of tasks results in fine-grained decomposition and that into a small number of tasks results in a coarse grained decomposition.



A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

Granularity, and Communication

- Finest granularity helps for a larger parallelism and to exploit different levels of parallelism
- But in general, if the granularity of a decomposition is finer, the associated overhead (as a ratio of useful work associated with a task) increases.

Minimizing Communication

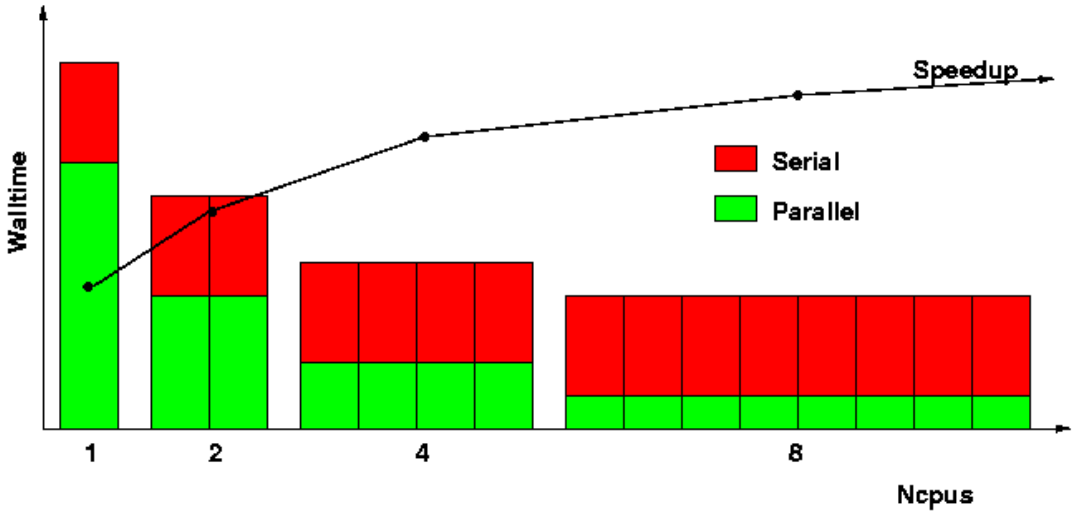
- When possible reduce the communication events:
- Group lots of small communications into large one.
- Eliminate synchronizations as much as possible. Each synchronization level off the performance to that of the slowest process.

Overlap Communication and Computation

- When possible code your program in such a way that processes continue to do useful work while communicating.
- This is usually a non trivial task and is afforded in the very last phase of parallelization.
- If you succeed, you have done. Benefits are enormous.

Amdahl's law

In a massively parallel context, an upper limit for the scalability of parallel applications is determined by the fraction of the overall execution time spent in non-scalable operations (Amdahl's law).



maximum speedup tends to
$1 / (1 - P)$
P= parallel fraction

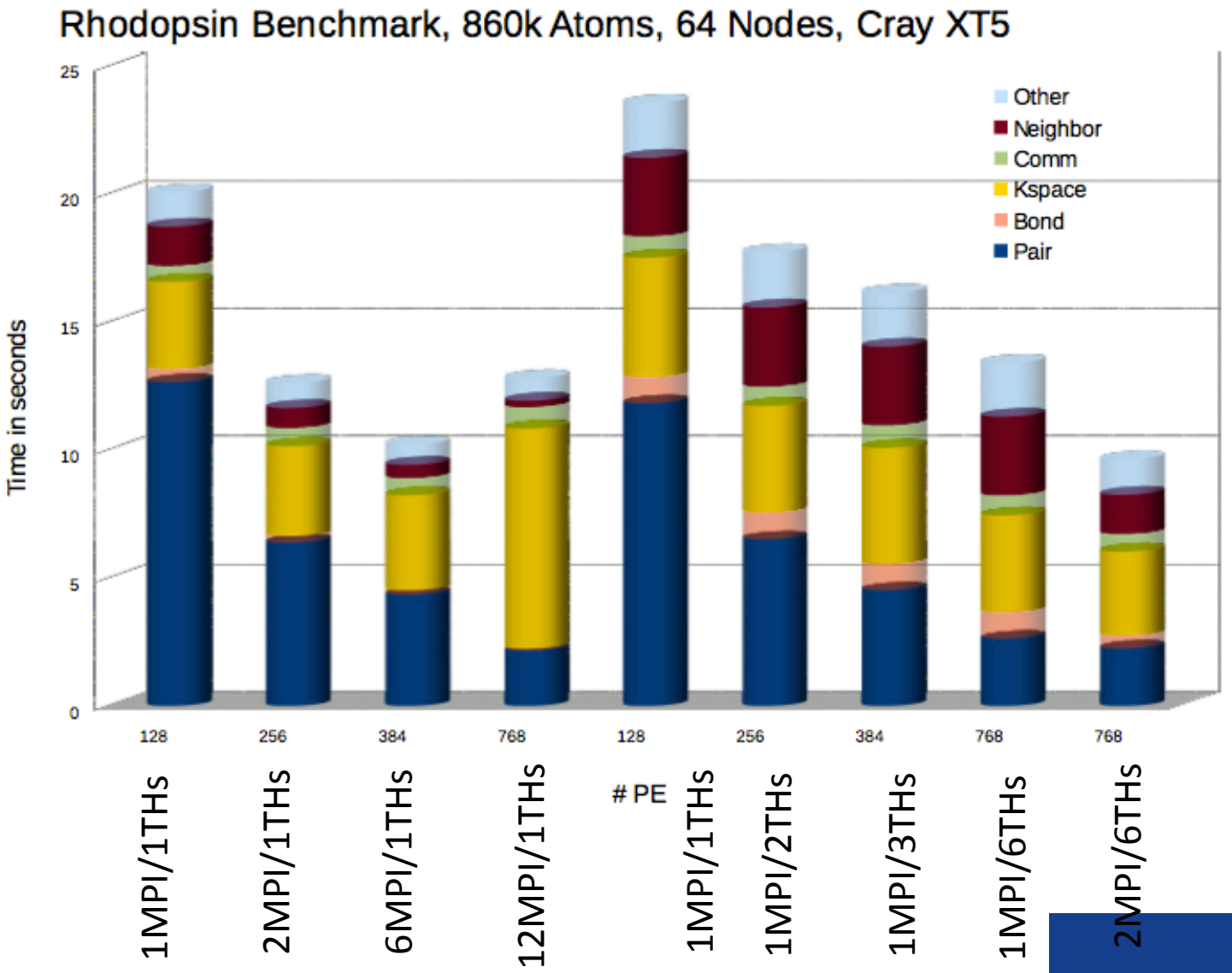
1000000 core

P = 0.999999

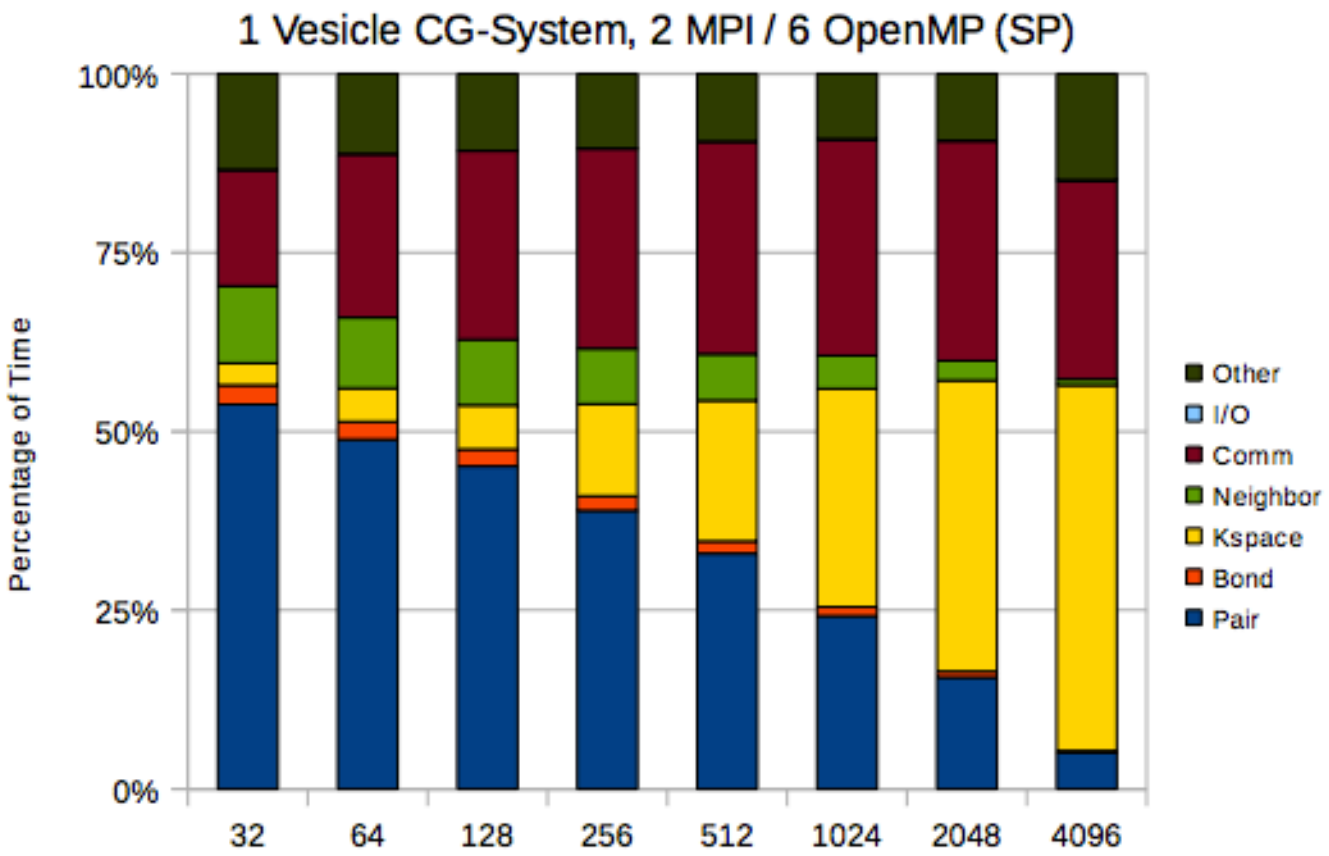
serial fraction= 0.000001

Scalability Limits

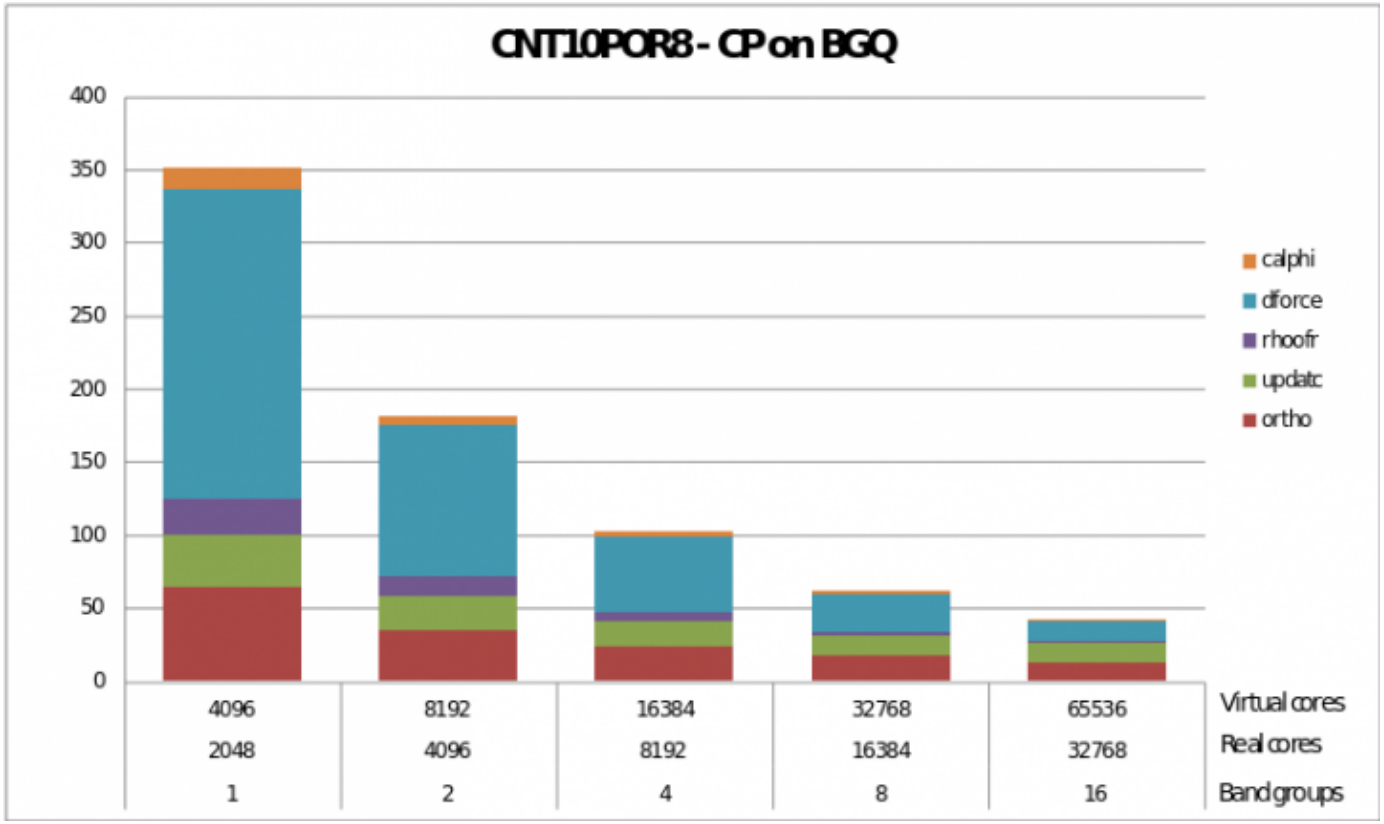
- Amdahl's law (there is a serial part which is not effected)
- Parallelism introduces overhead due to communication, synchronization, additional operations (I/O, memory copies, reordering, etc...)
- Sometimes the limit of scalability can be somehow predicted:
  - is there enough work for any process?


Rhodopsin Benchmark, 860k Atoms, 64 Nodes, Cray XT5
Time in seconds
0
5
10
15
20
25
Other
Neighbor
Comm
Kspace
Bond
Pair
128
256
384
768
128
256
384
768
768
PE
1MPI/1THs
2MPI/1THs
6MPI/1THs
12MPI/1THs
1MPI/1THs
1MPI/2THs
1MPI/3THs
1MPI/6THs
2MPI/6THs

Amdal's Law And Real Life



Scaling - QE-CP on Fermi BGQ @ CINECA



References

- MPI Documentation
- MPI APIs (list) Documentation

Thanks for your attention!!