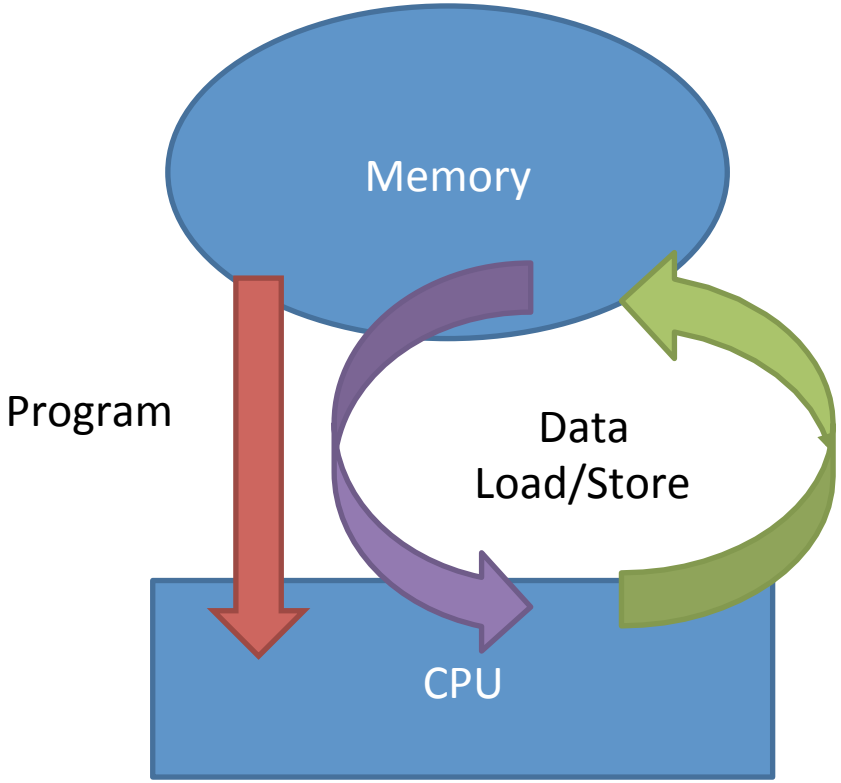
# Overview of Common Strategies for Parallelization

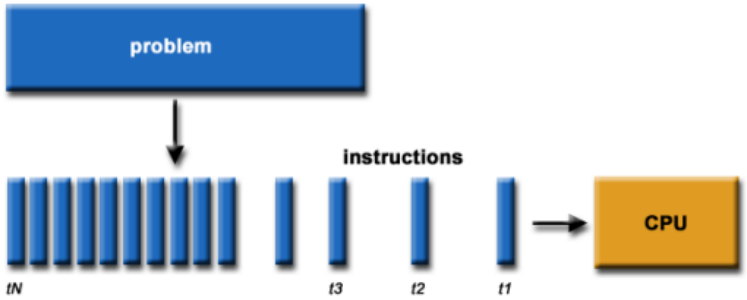
**Ivan Girotto – igirotto@ictp.it**

International Centre for Theoretical Physics (ICTP)

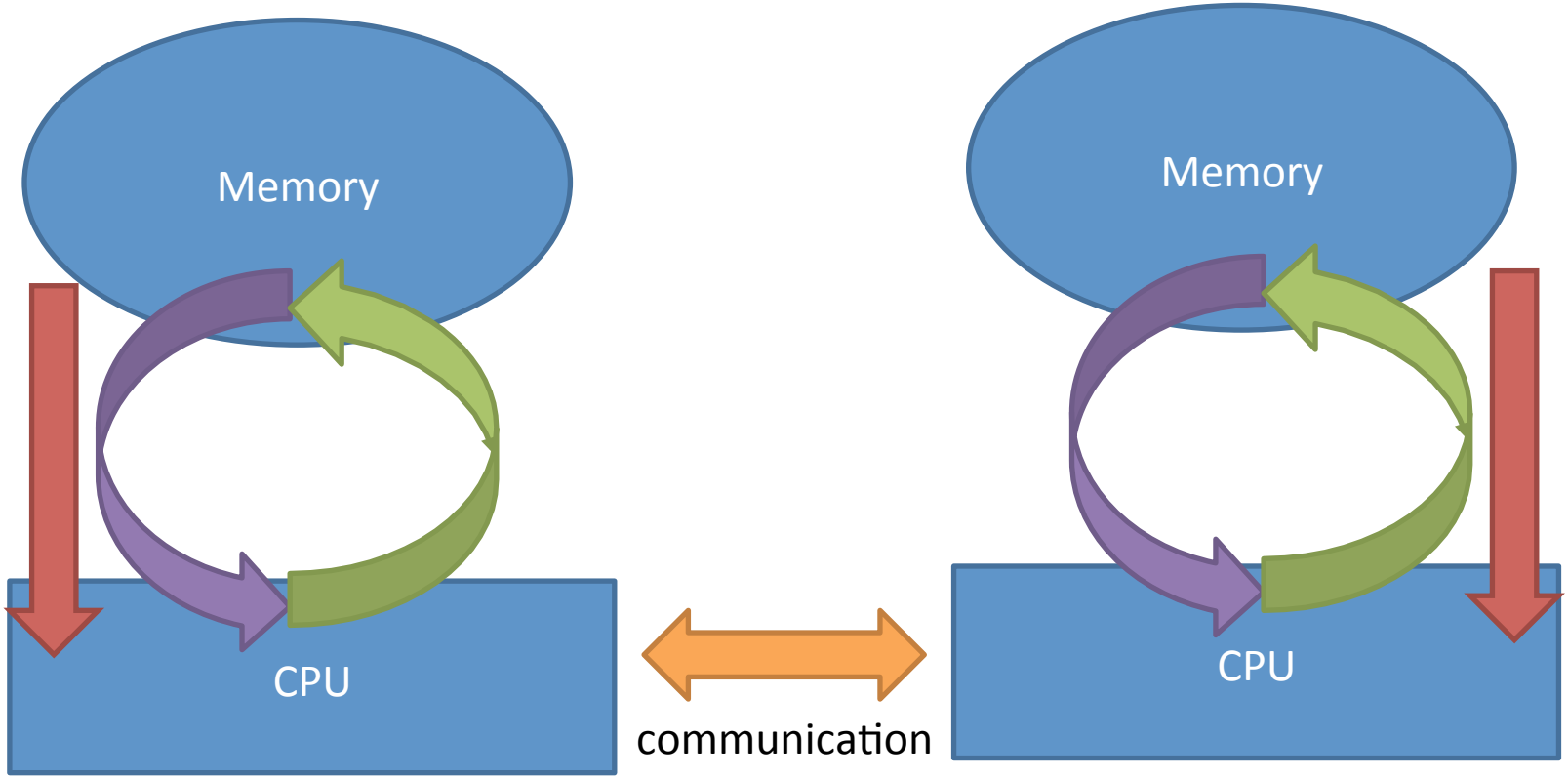
# Serial Programming




A problem is broken into a discrete series of instructions.
Instructions are executed one after another.
Only one instruction may execute at any moment in time.

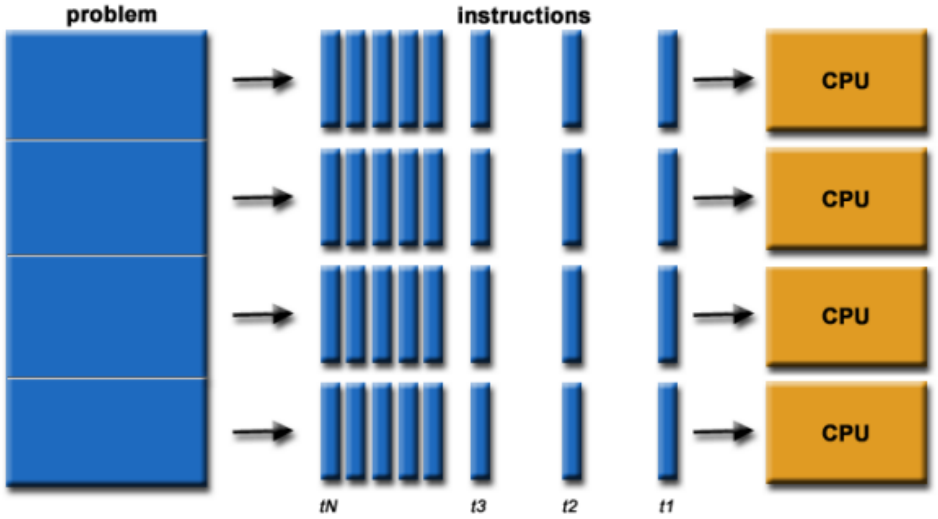


# Parallel Programming



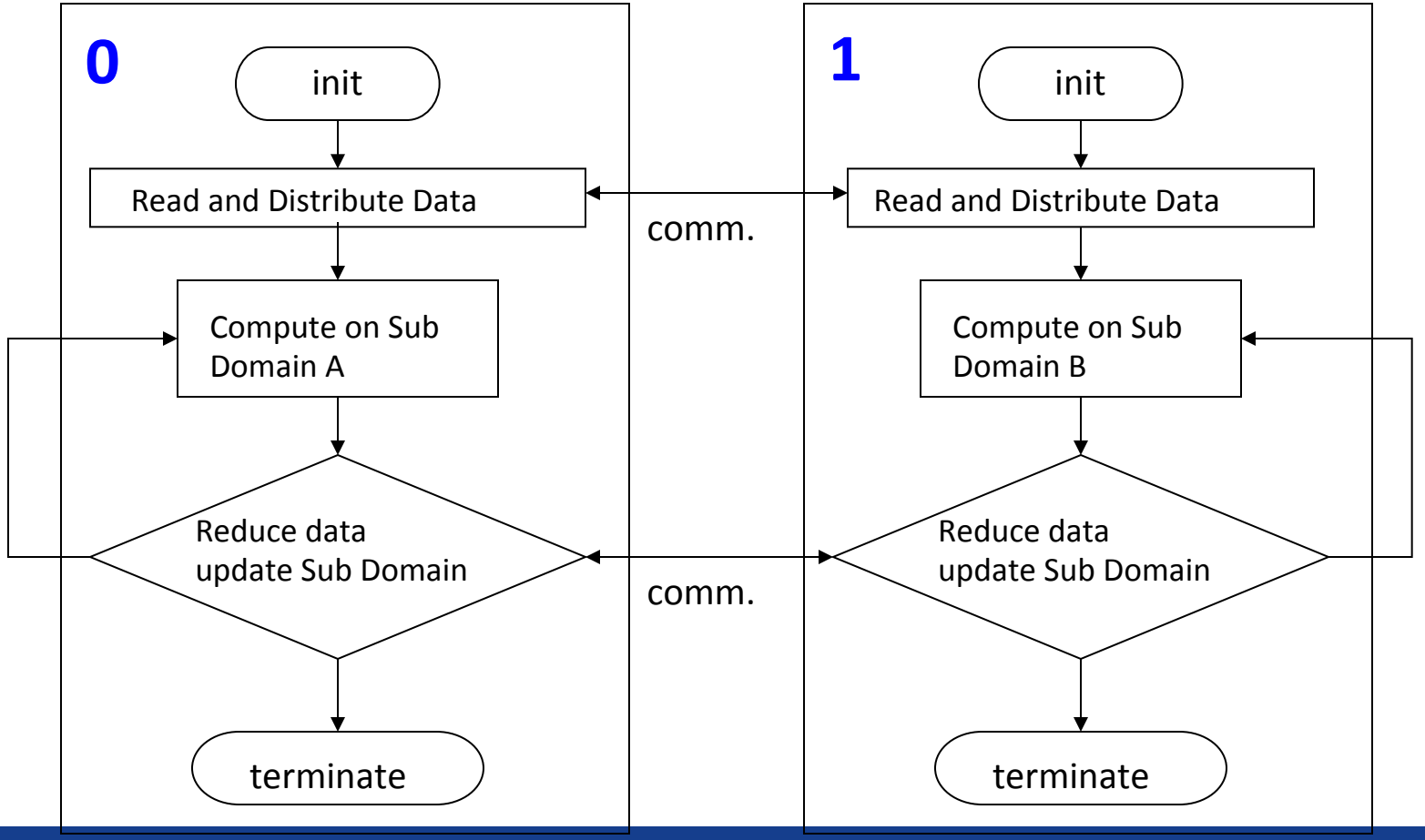
# Concurrency

The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently




- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control / coordination mechanism is employed

# What is a Parallel Program

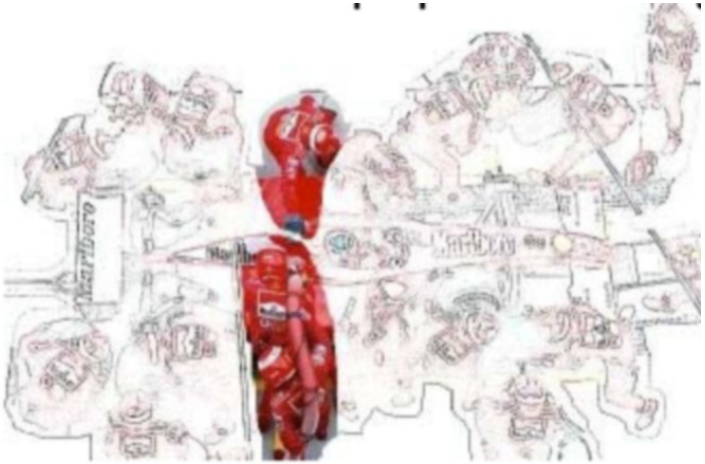


# Fundamental Steps of Parallel Design

- Identify portions of the work that can be performed concurrently
- Mapping the concurrent pieces of work onto multiple processes running in parallel
- Distributing the input, output and intermediate data associated within the program
- Managing accesses to data shared by multiple processors
- Synchronizing the processors at various stages of the parallel program execution

# Type of Parallelism

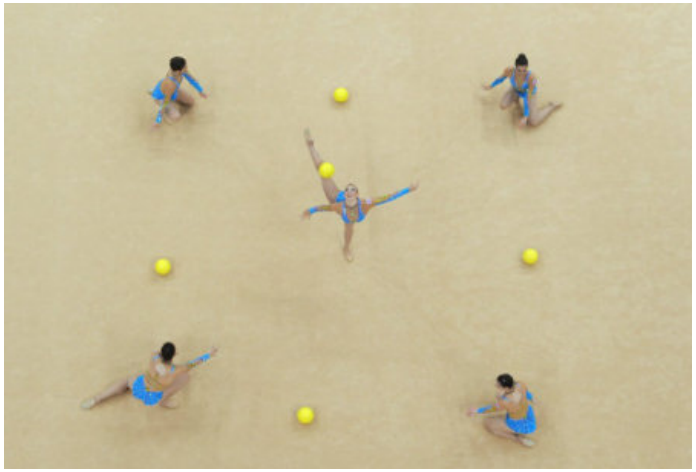
- **Functional (or task) parallelism**: different people are performing different task at the same time



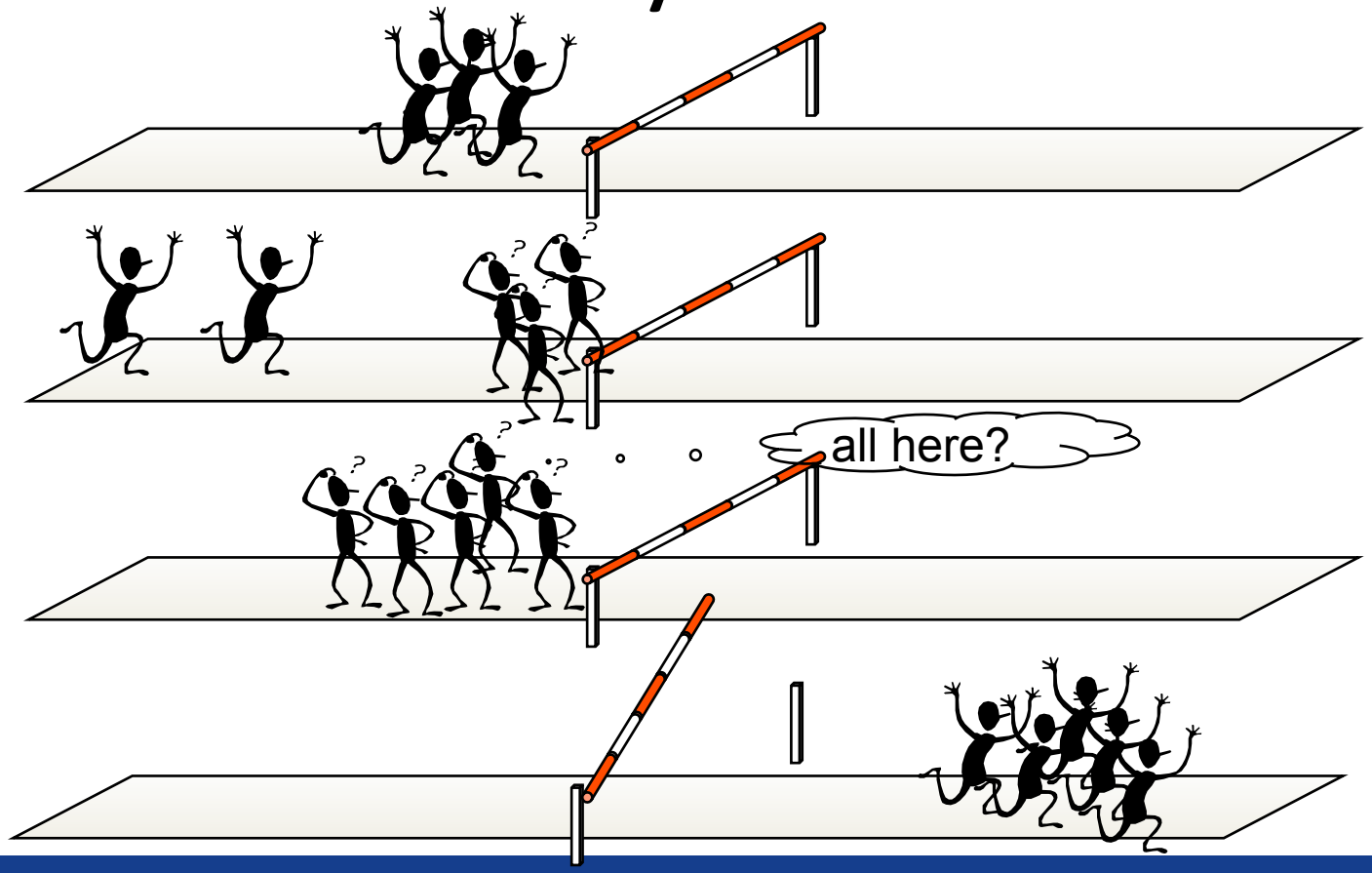
- **Data Parallelism**: different people are performing the same task, but on different equivalent and independent objects

# Process Interactions

- The effective speed-up obtained by the parallelization depend by the amount of overhead we introduce making the algorithm parallel
- There are mainly two key sources of overhead:
  1. Time spent in inter-process interactions (**communication**)
  2. Time some process may spent being idle (**synchronization**)

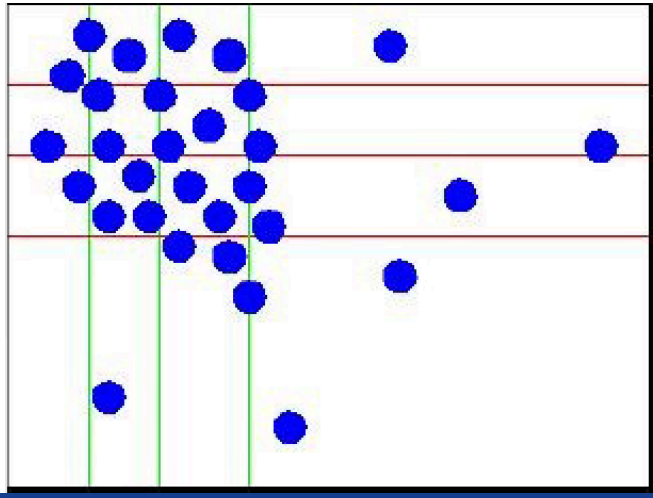


# Barrier and Synchronization



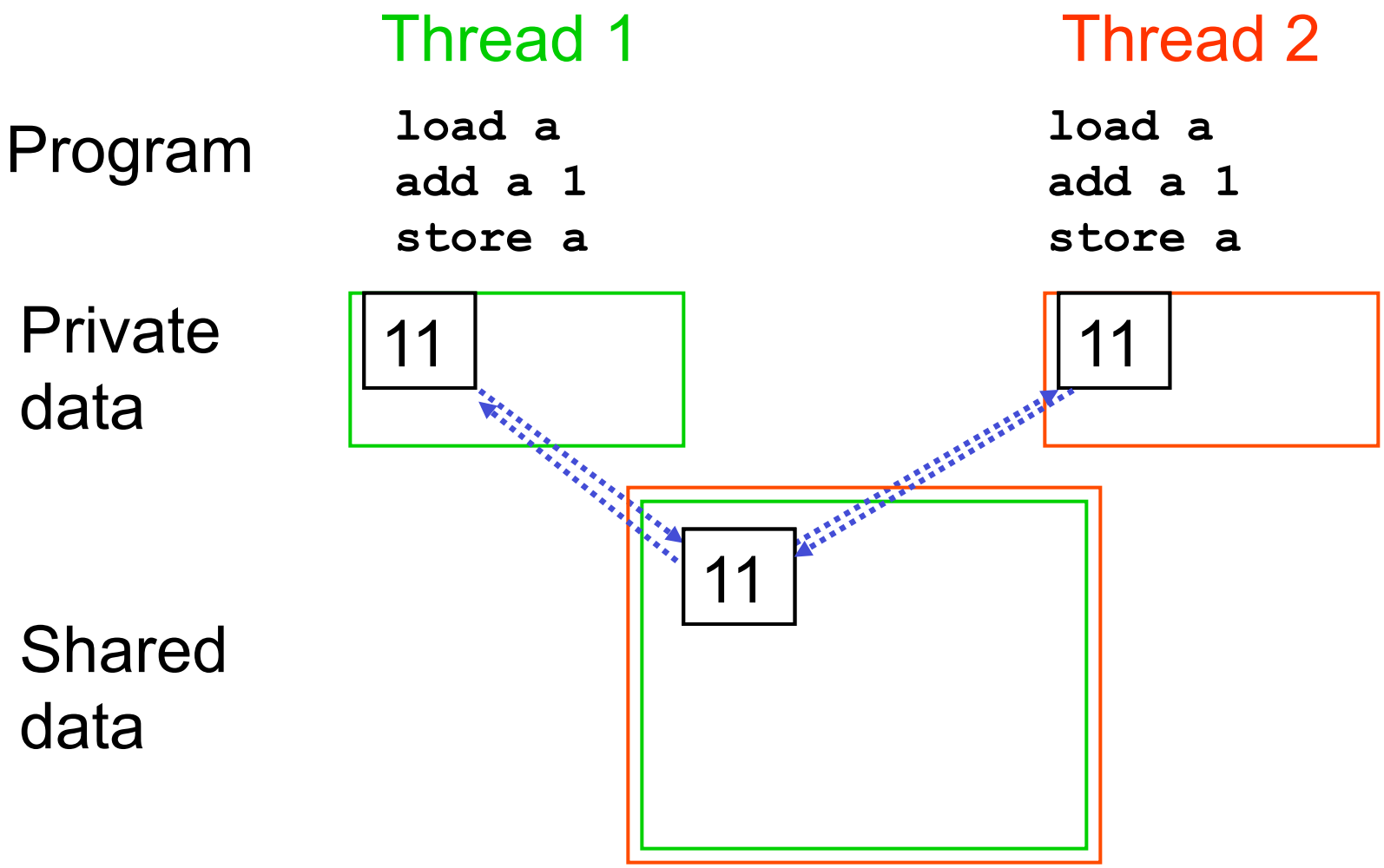
# Limitations of Parallel Computing

- Fraction of serial code limits parallel speedup
- Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution
- Load imbalance:
  - parallel tasks have a different amount of work
  - CPUs are partially idle
  - redistributing work helps but has limitations
  - communication and synchronization overhead



# Shared Resources

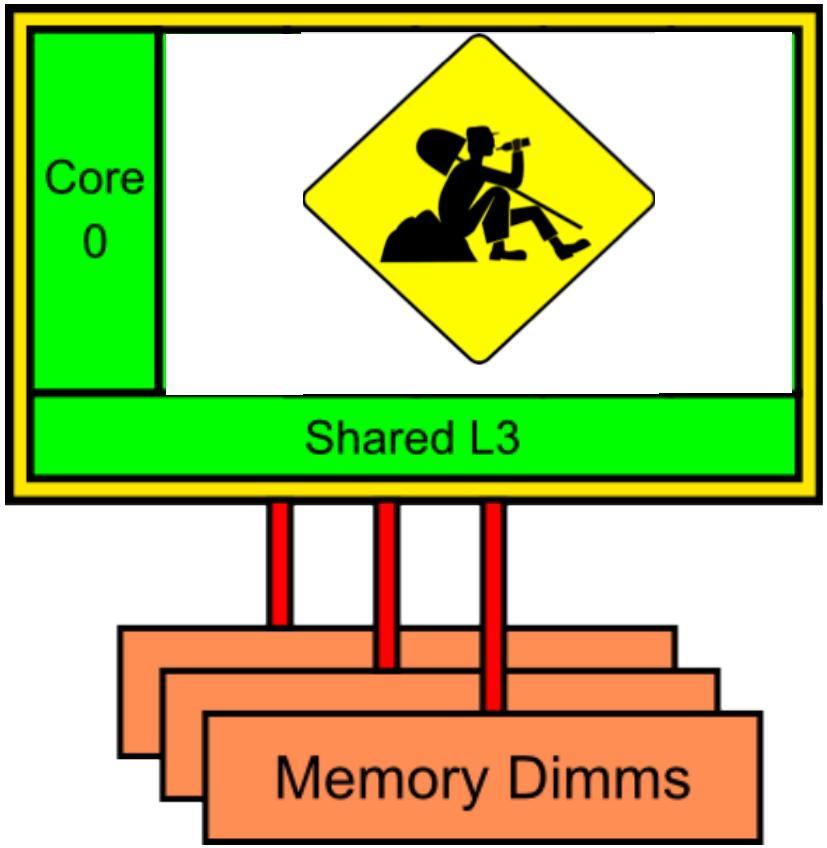
- In parallel programming, developers must manage exclusive access to shared resources
- Resources are in different forms:
  - concurrent read/write (including parallel write) to shared memory locations
  - concurrent read/write (including parallel write) to shared devices
  - a message that must be send and received

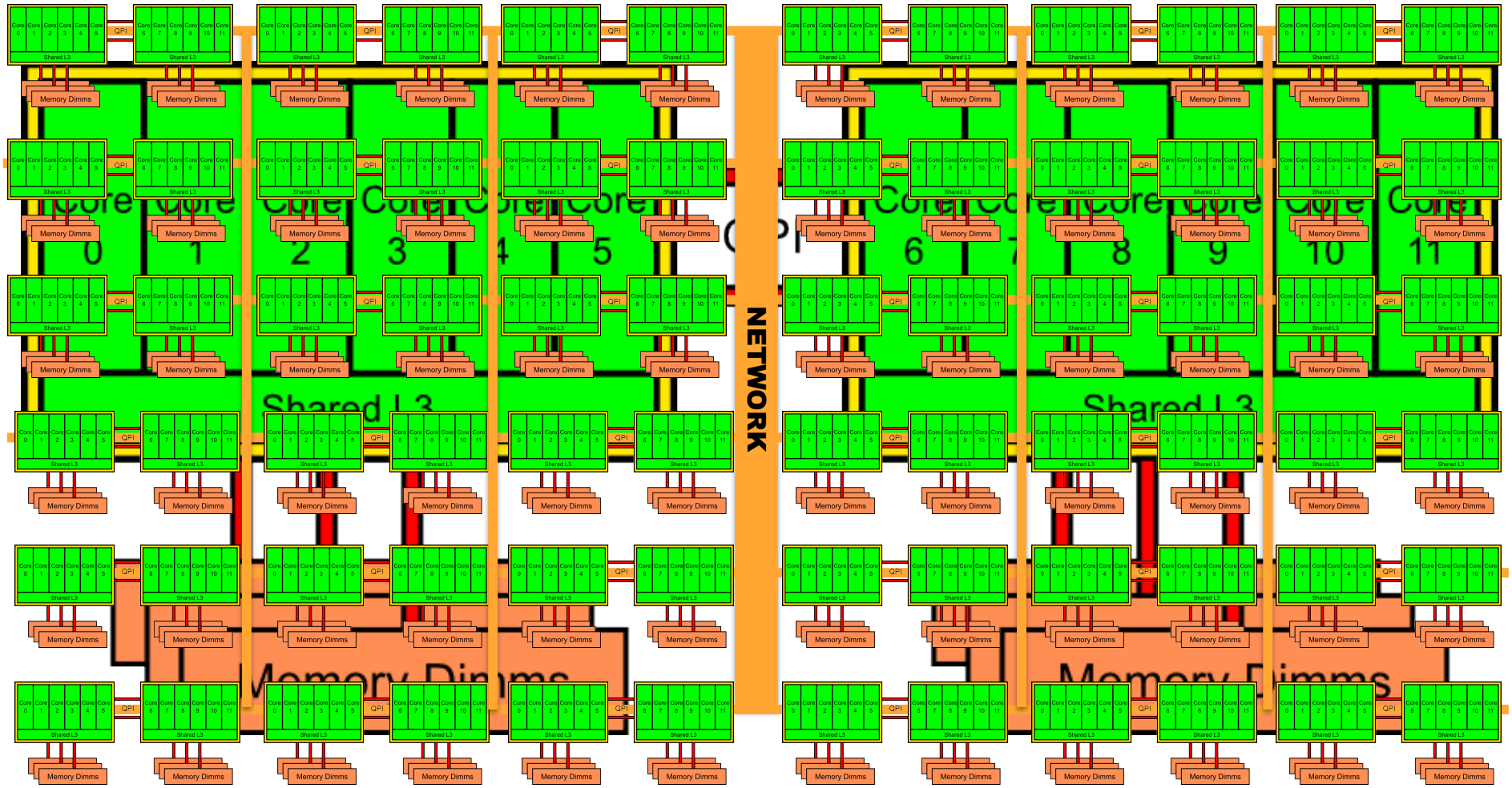

Thread 1
Thread 2
Program
load a
add a 1
store a
load a
add a 1
store a
Private data
11
11
Shared data
11

# 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)

# Scalable Programming




QPI
Shared L3
Memory Dimms
NETWORK

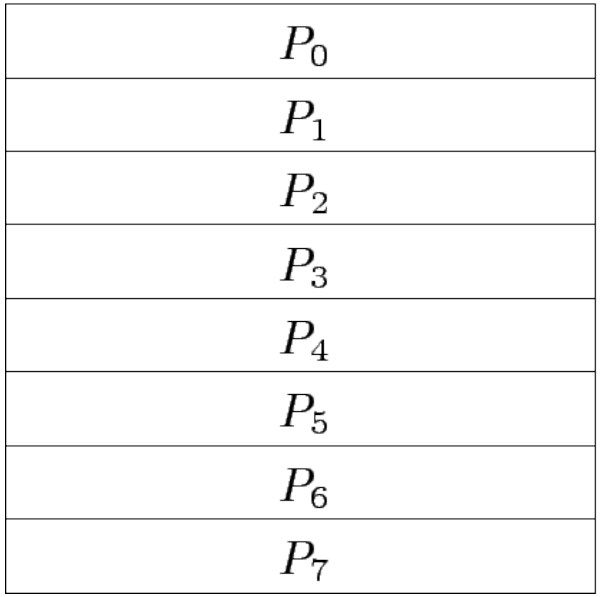
# Granularity

- Granularity is determined by the decomposition level (number of task) on which we want divide the problem
- The degree to which task/data can be subdivided is limit to concurrency and parallel execution
- Parallelization has to become “topology aware”
  - coarse grain and fine grained parallelization has to be mapped to the topology to reduce memory and I/O contention
  - make your code modularized to enhance different levels of granularity and consequently to become more “platform adaptable”

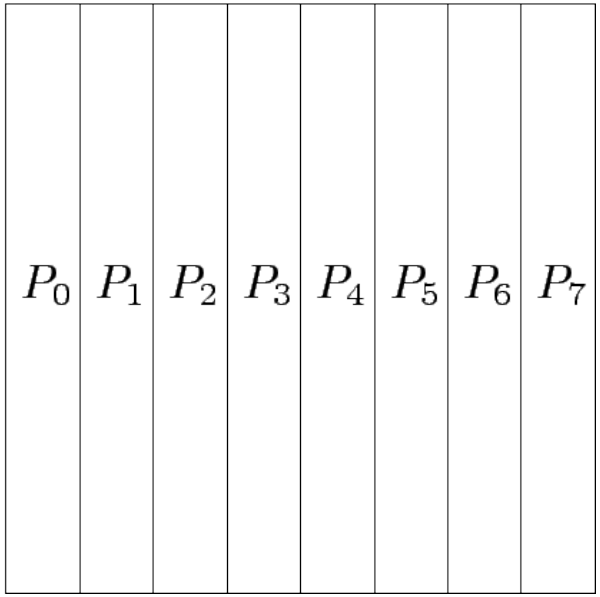
# Static Data Partitioning

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

row-wise distribution

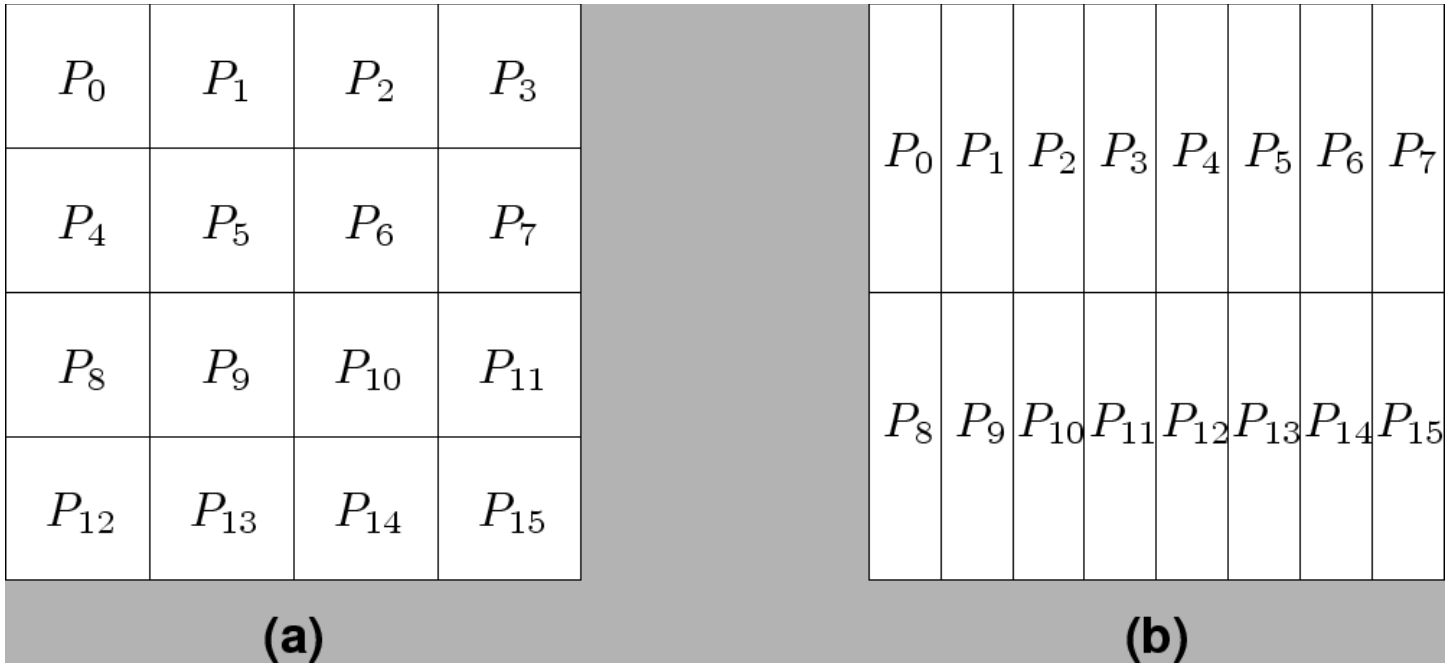



column-wise distribution



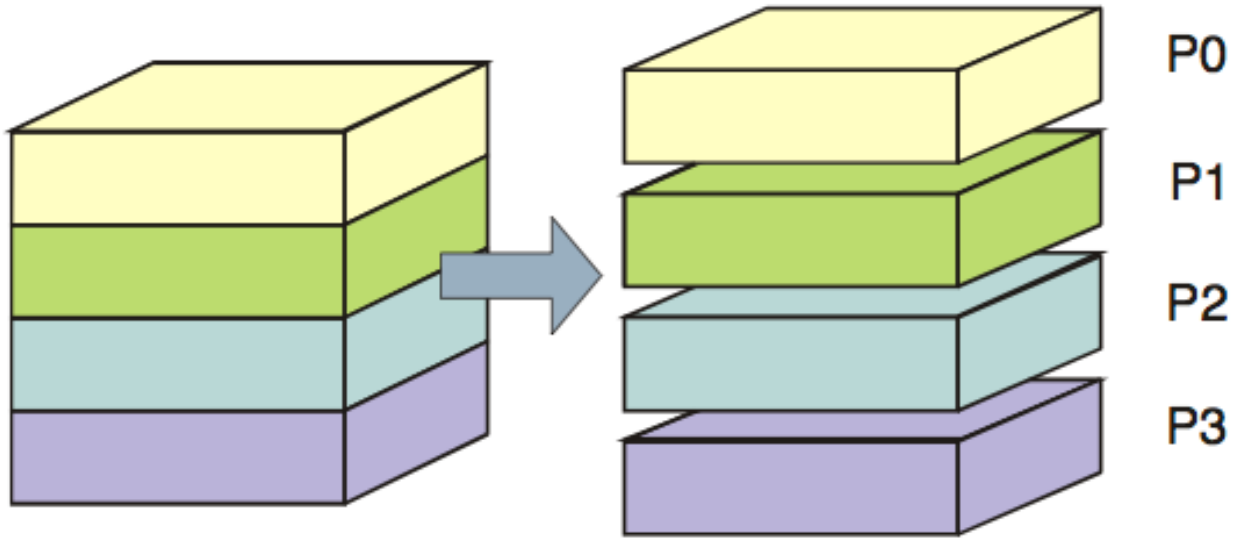
# Block Array Distribution Schemes

**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!!**

# 1D Distribution of a 3D domain

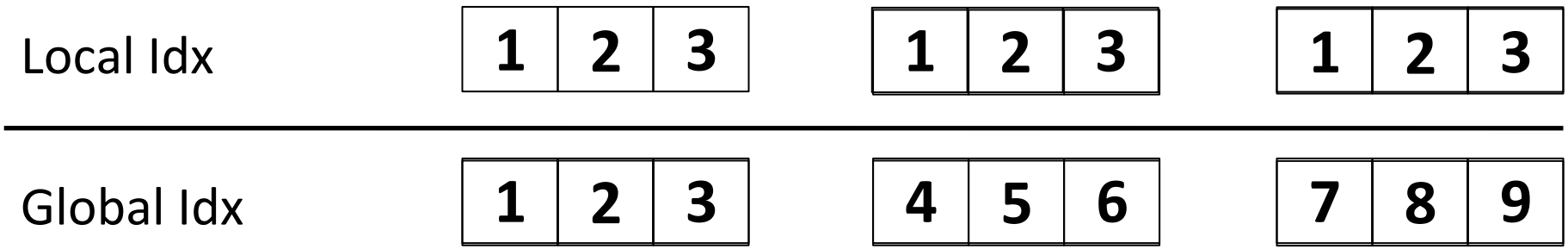


# Distributed Data Vs Replicated Data

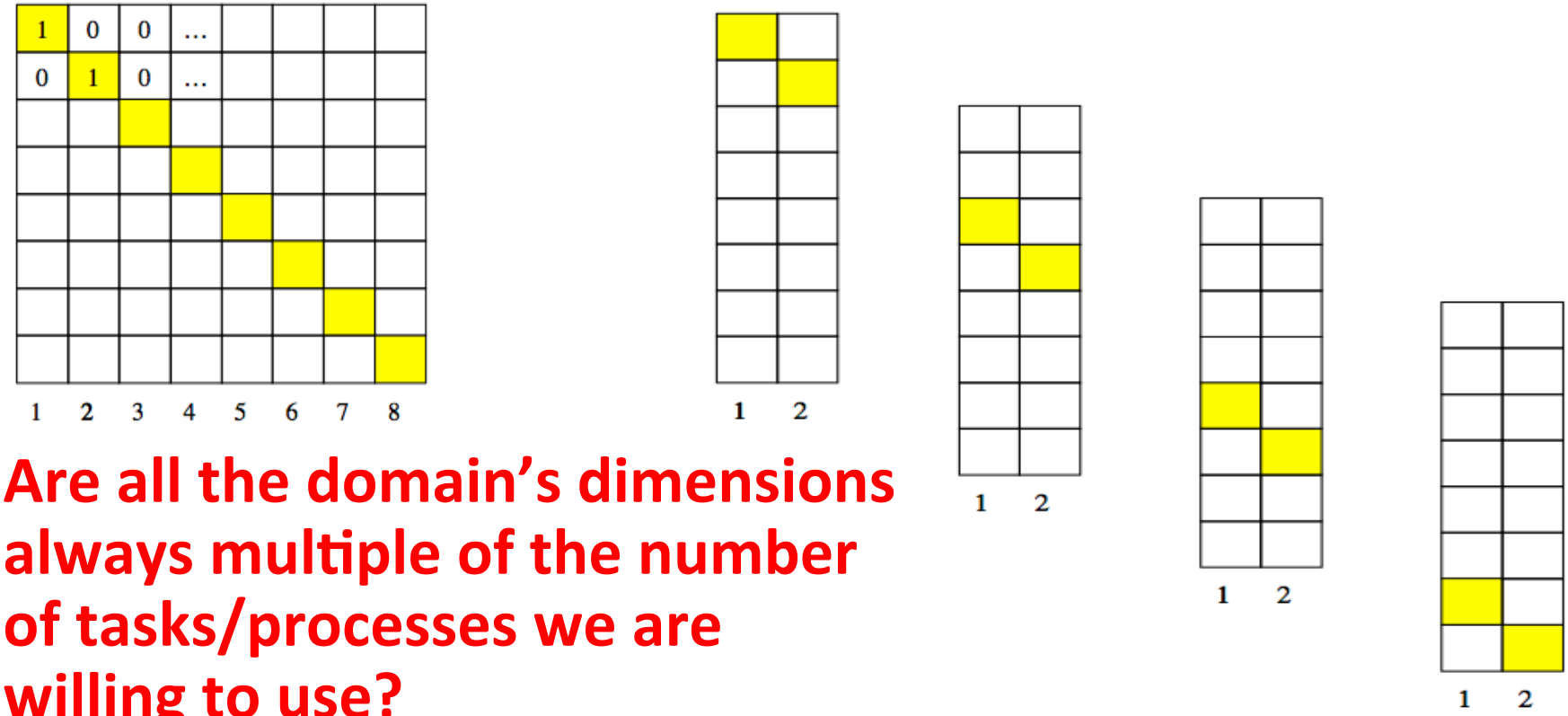
- Replicated data distribution is useful if it helps to reduce the communication among process at the cost of bounding scalability
- Distributed data is the ideal data distribution but not always applicable for all data-sets
- Usually complex application are a mix of those techniques => distribute large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



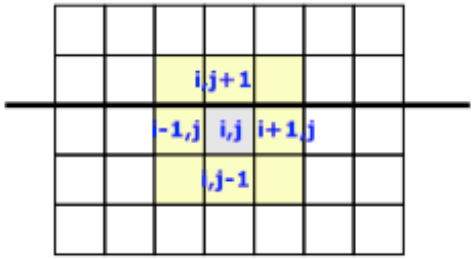
# Collaterals to Domain Decomposition /1



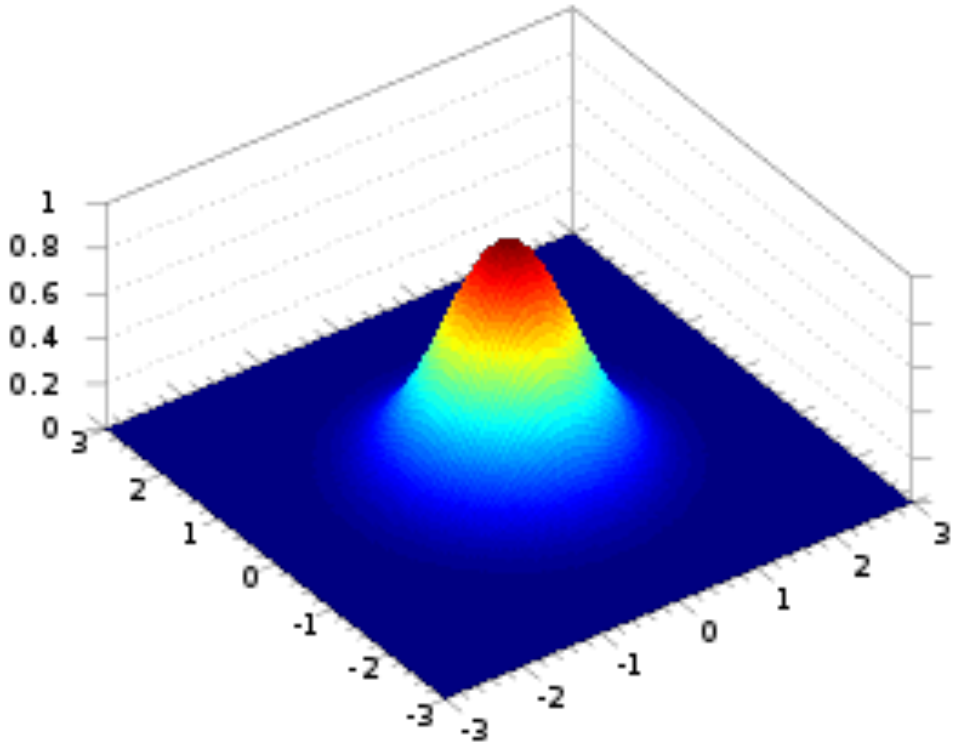

**Are all the domain's dimensions always multiple of the number of tasks/processes we are willing to use?**

# Again on Domain Decomposition

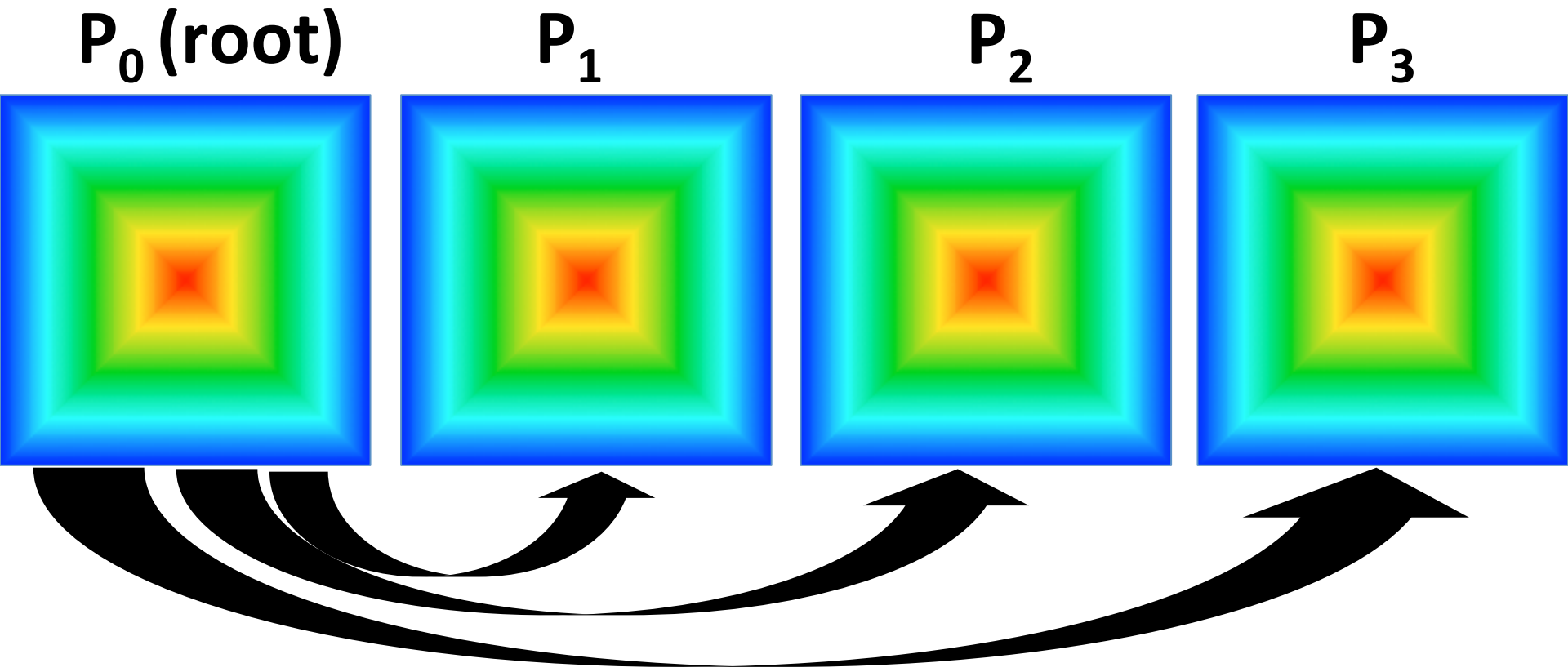
sub-domain boundaries

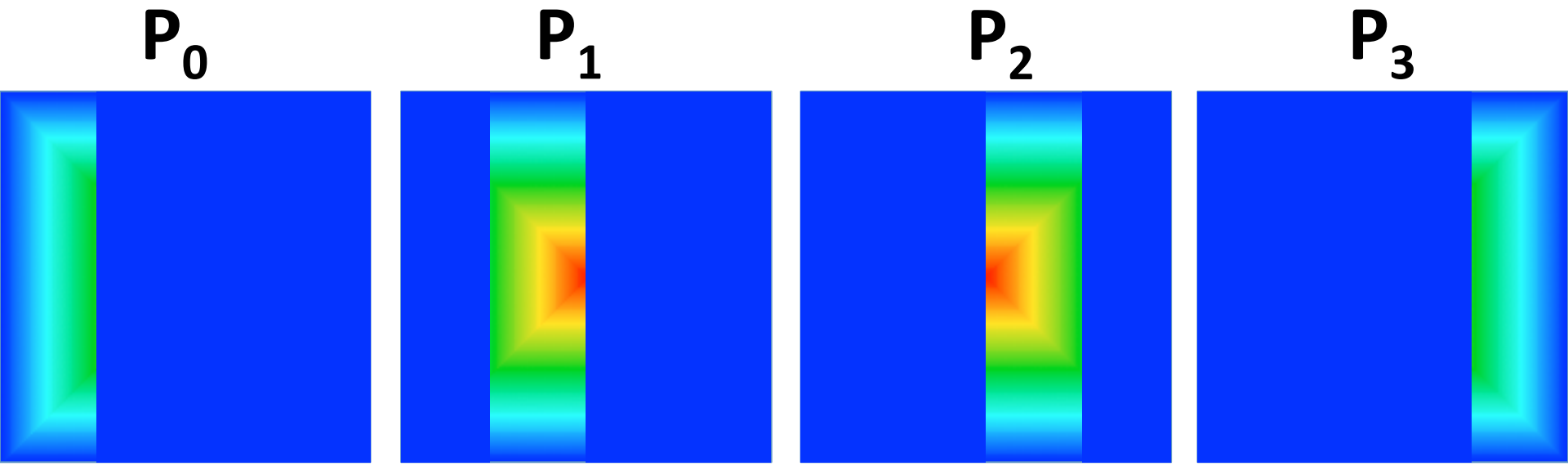


$P_0$



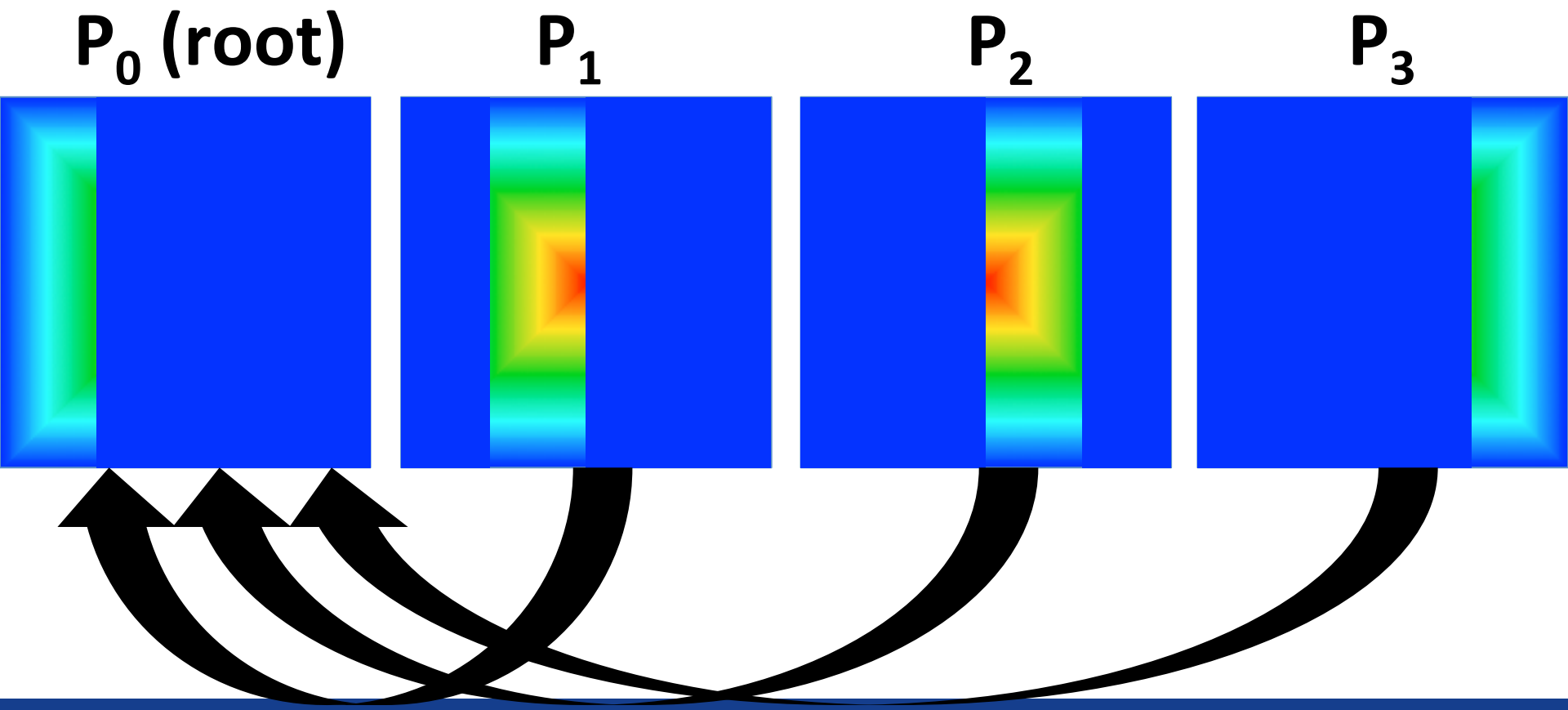

# call MPI_BCAST( ... )






**call evolve( dtfact )**

# call MPI_Gather( …, …, … )

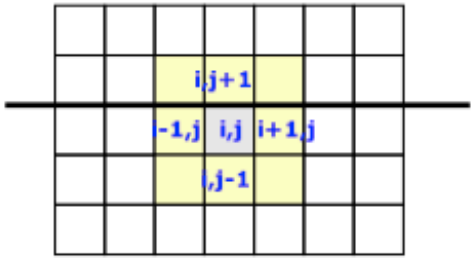


# Replicated data

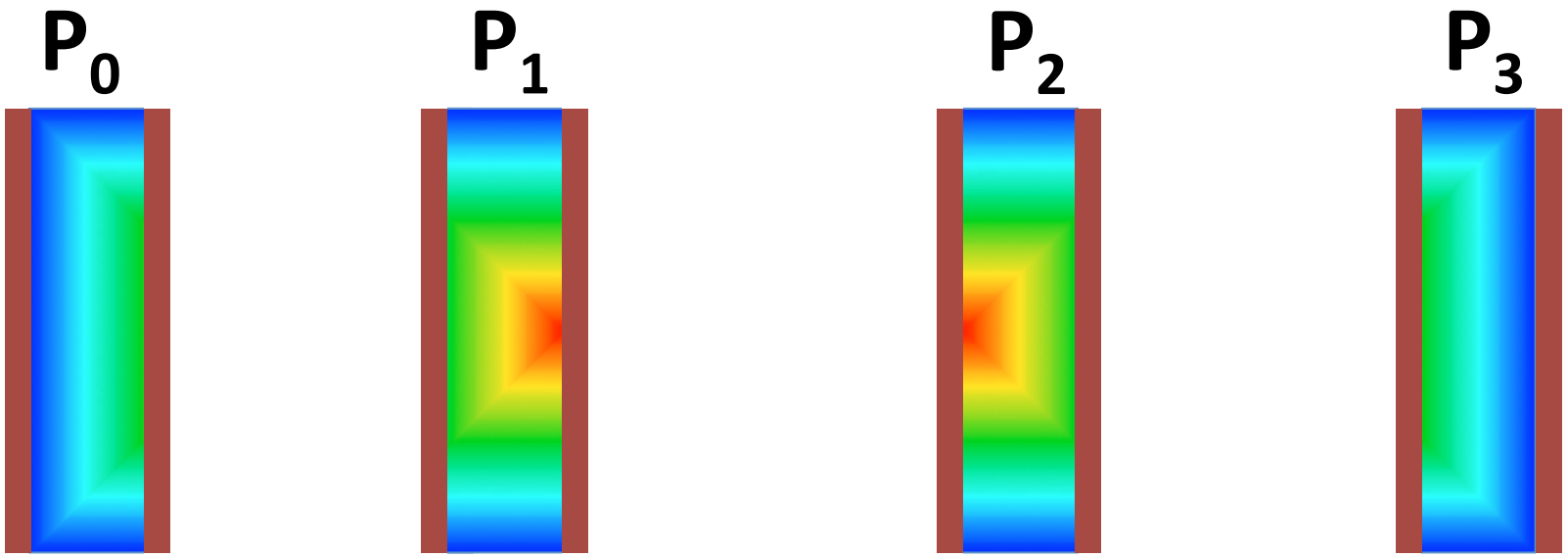
- Compute domain (and workload) distribution among processes
- Master-slaves: $P_0$ drives all processes
- Large amount of data communication
  - at each step $P_0$ distribute data to all processes and collect the contribution of each process
- Problem size scaling limited in memory capacity

# Collaterals to Domain Decomposition /2

sub-domain boundaries

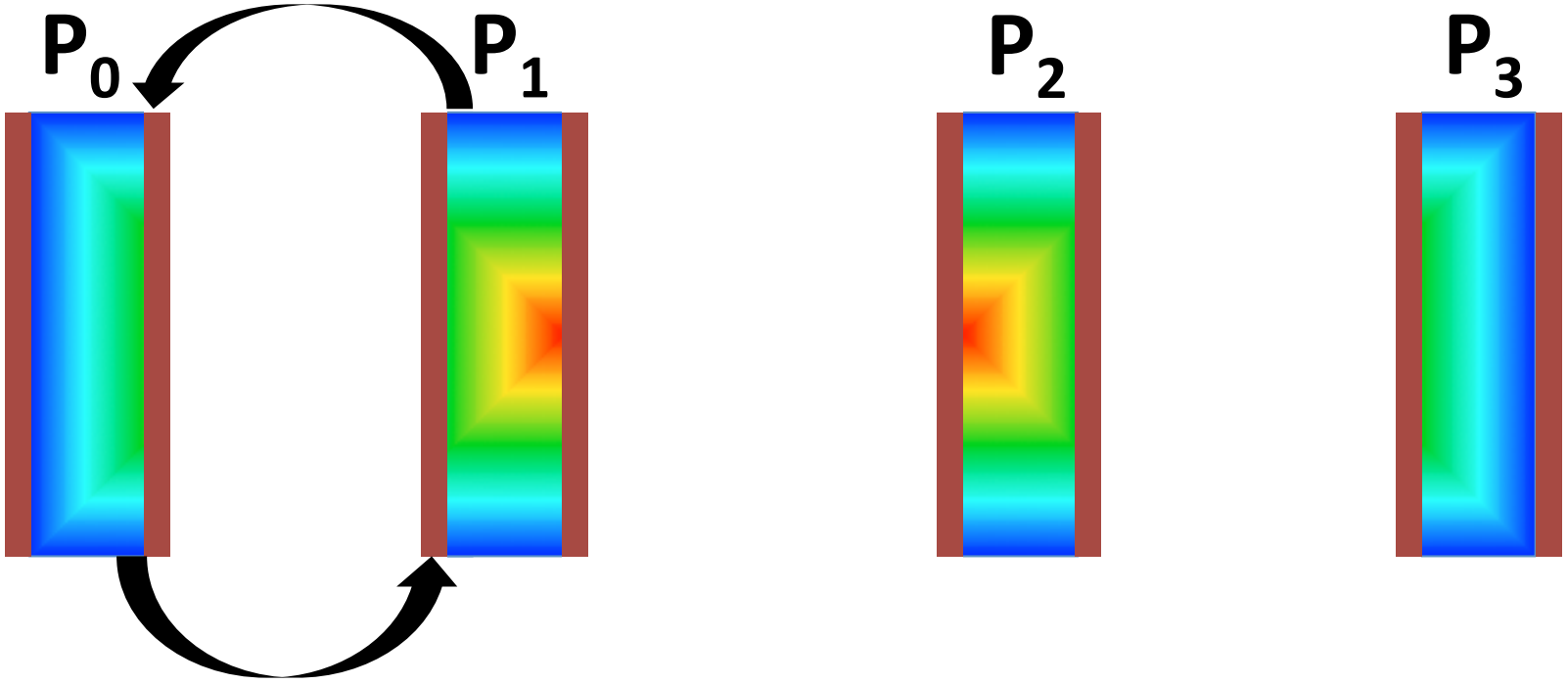


# The Transport Code - Parallel Version

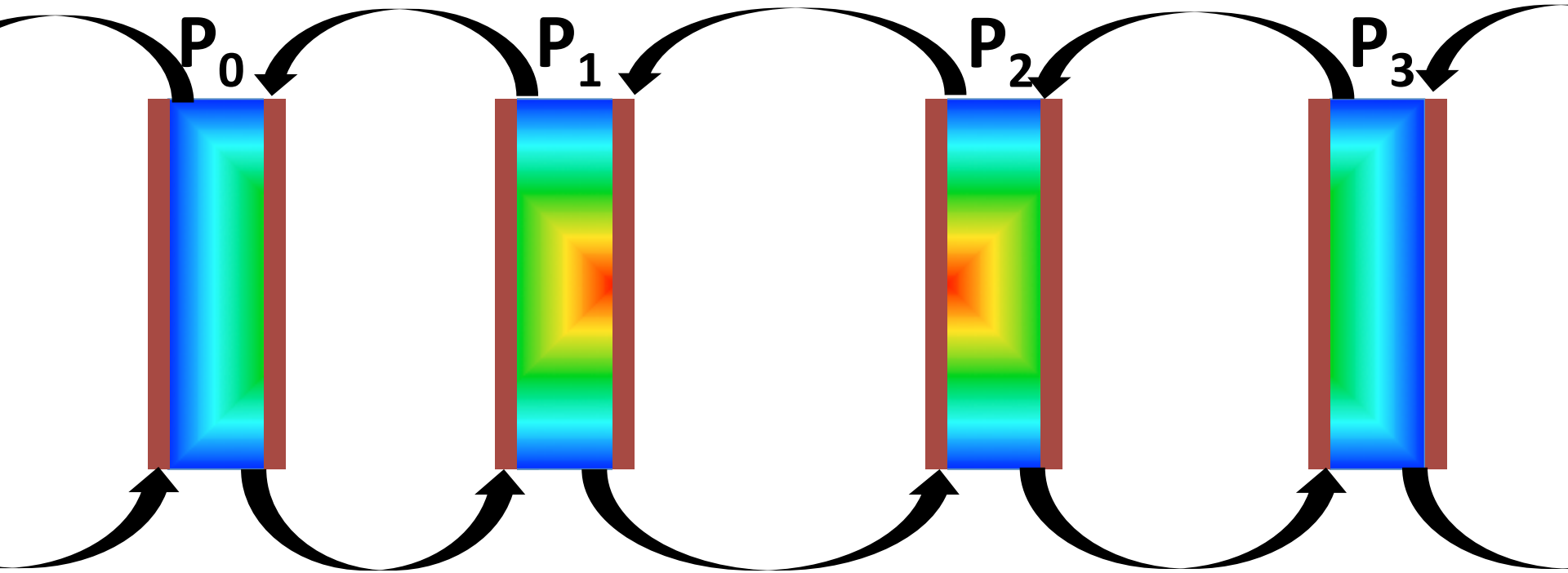



**call evolve( dtfact )**

# Data exchange among processes



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




proc_up = mod(proc_me + 1 , nprocs)

# Sendrecv

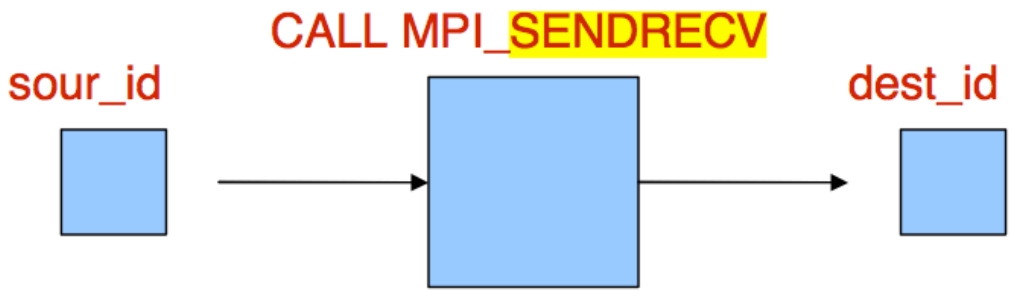
The easiest way to send and receive data without warring about deadlocks

Sender side

Fortran:

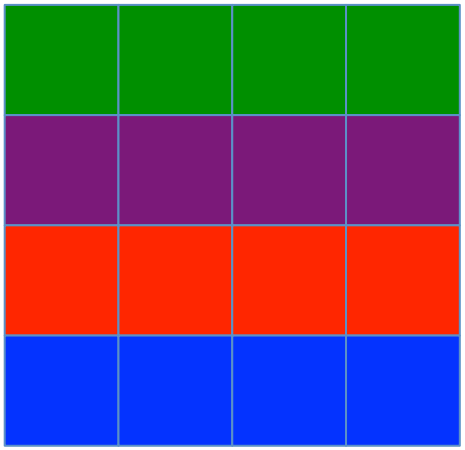
```
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type, dest_id, tag,
rcvbuf, rcv_size, rcv_type, sour_id, tag, comm, status, ierr)
```

Receiver side

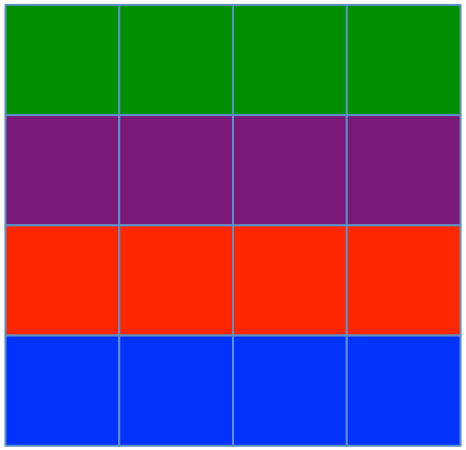


# Distributed Data

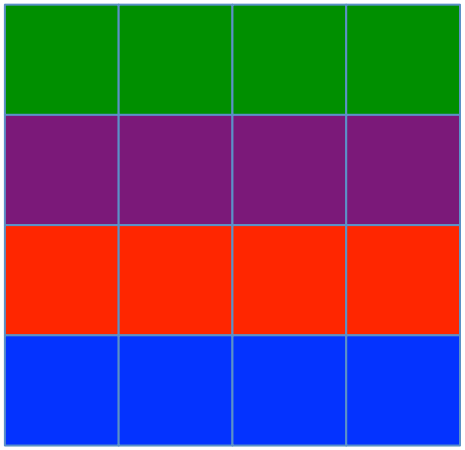
- Global and Local Indexes
- Ghost Cells Exchange Between Processes
  - Compute Neighbor Processes
- Parallel Output



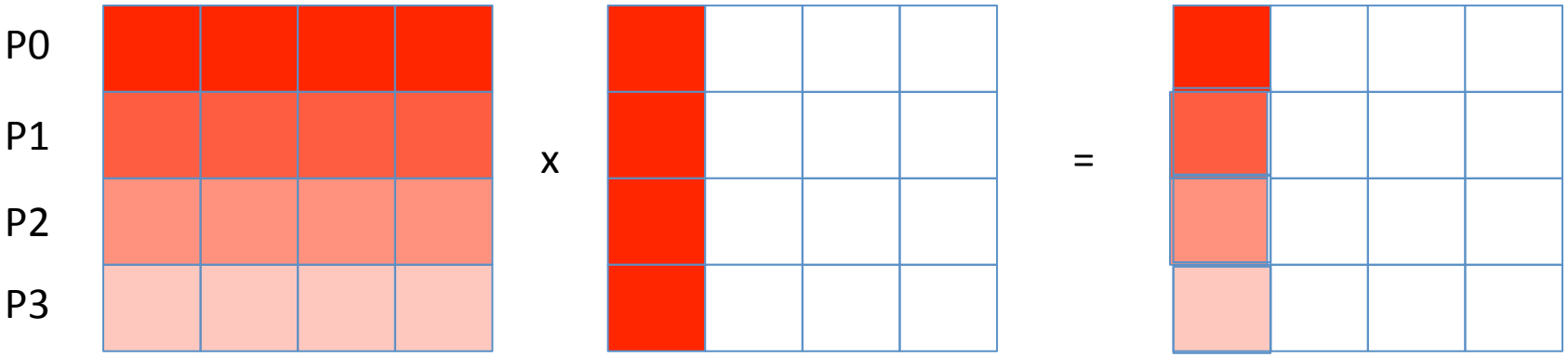
A



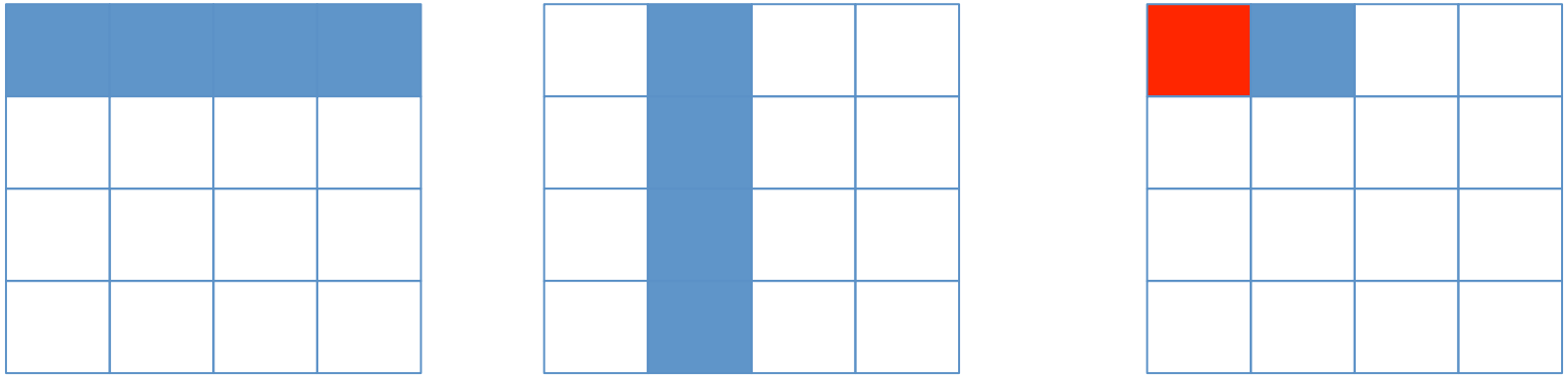
B

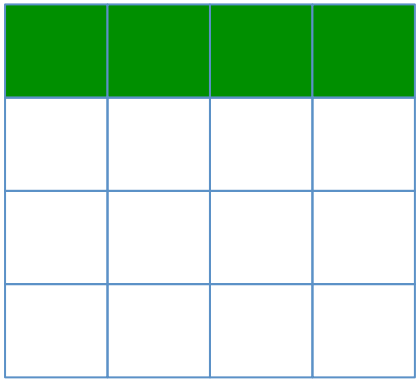
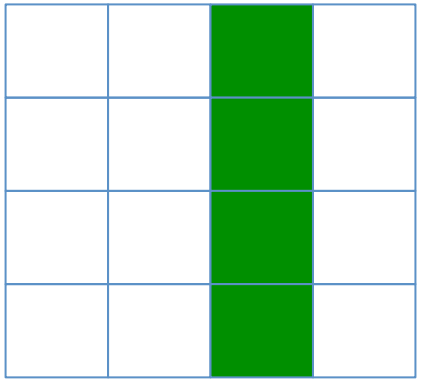
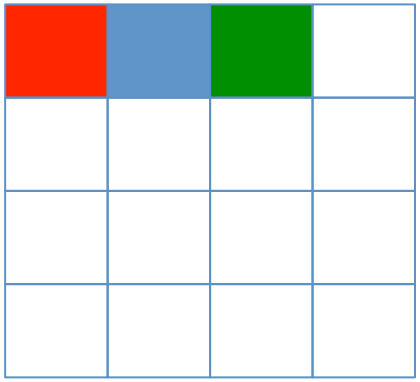
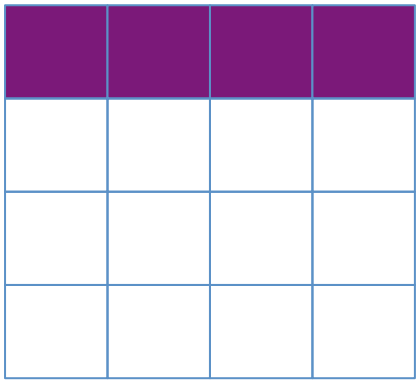
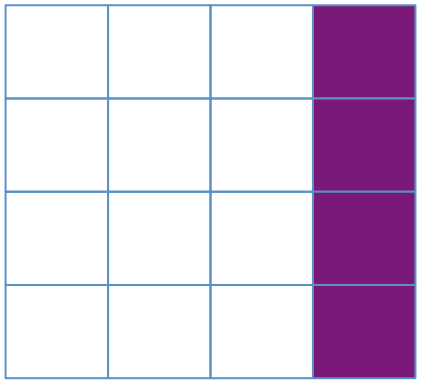


C




**At every step all the processes receive a block of columns of the Matrix B**




# MPI Allgather

```
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

**sendbuf** starting address of send buffer (choice)

**sendcount** number of elements in send buffer (integer)

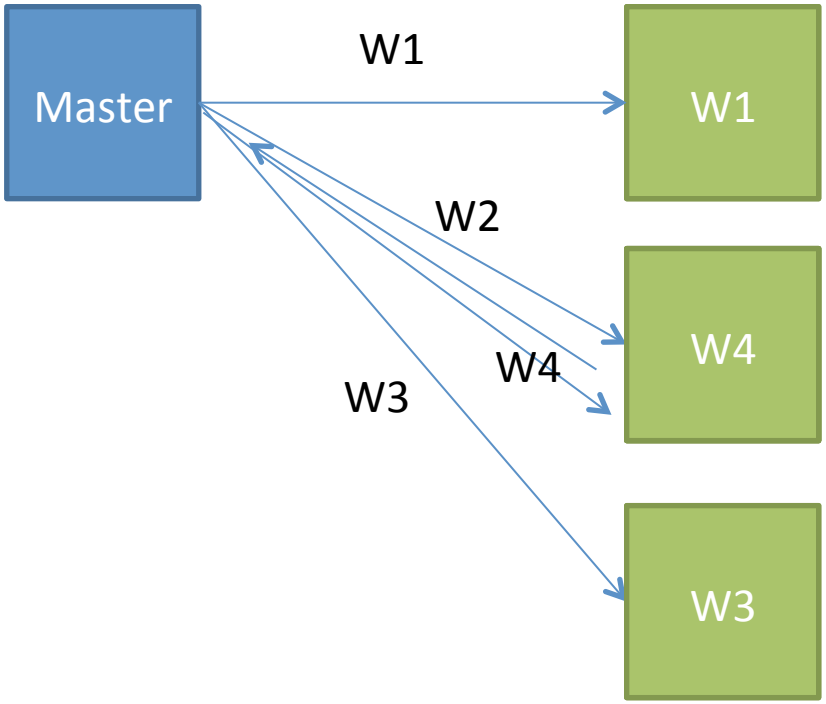
**sendtype** data type of send buffer elements (handle)

**recvcount** number of elements received from any process (integer)

**recvtype** data type of receive buffer elements (handle)

**comm** communicator (handle)

# 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

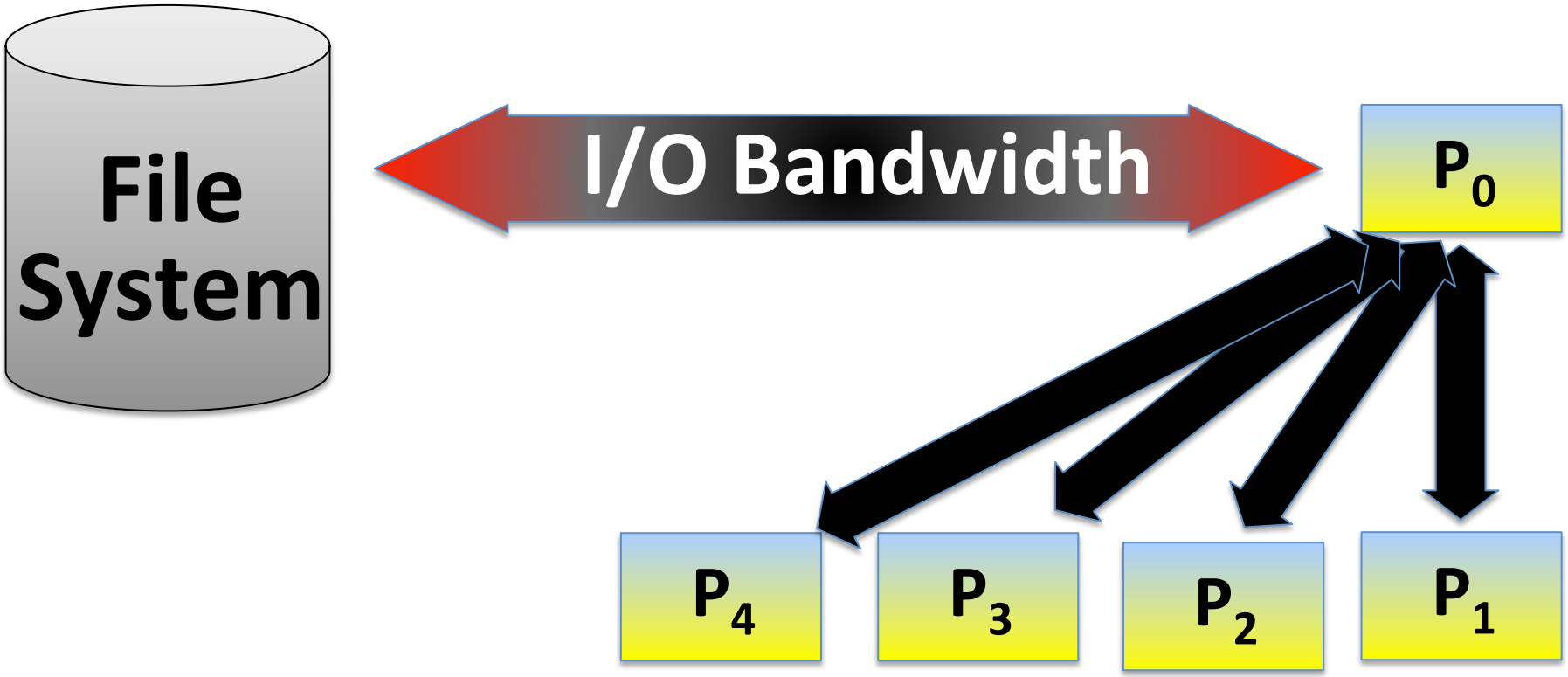
```
[igirotto@localhost]$ more my_shell_wrapper.sh
#!/bin/bash
#example for the OpenMPI implementation
./prog.x --input input_${OMPI_COMM_WORLD_RANK}.dat

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

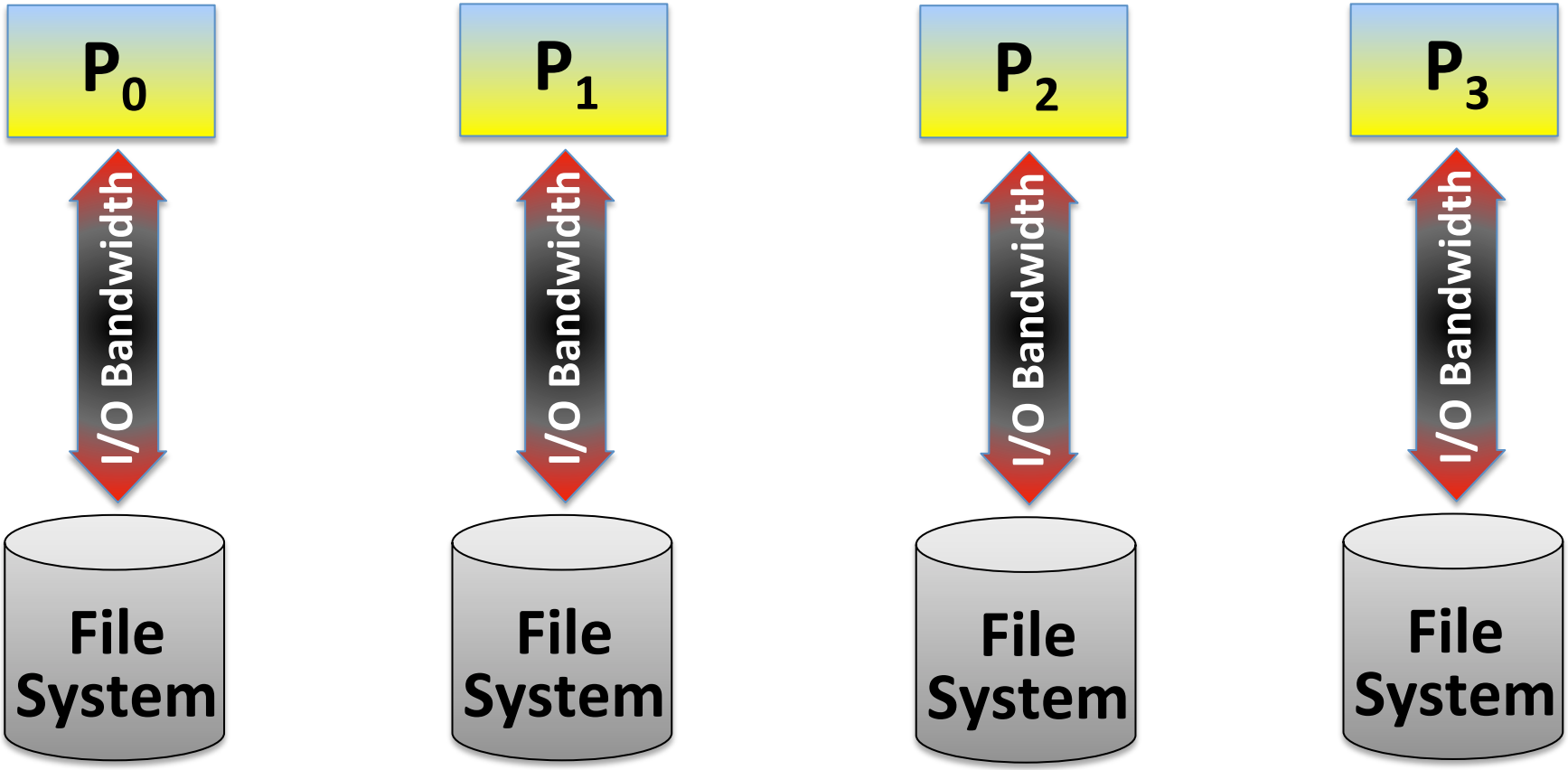
# Easy Parallel Computing

- 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)

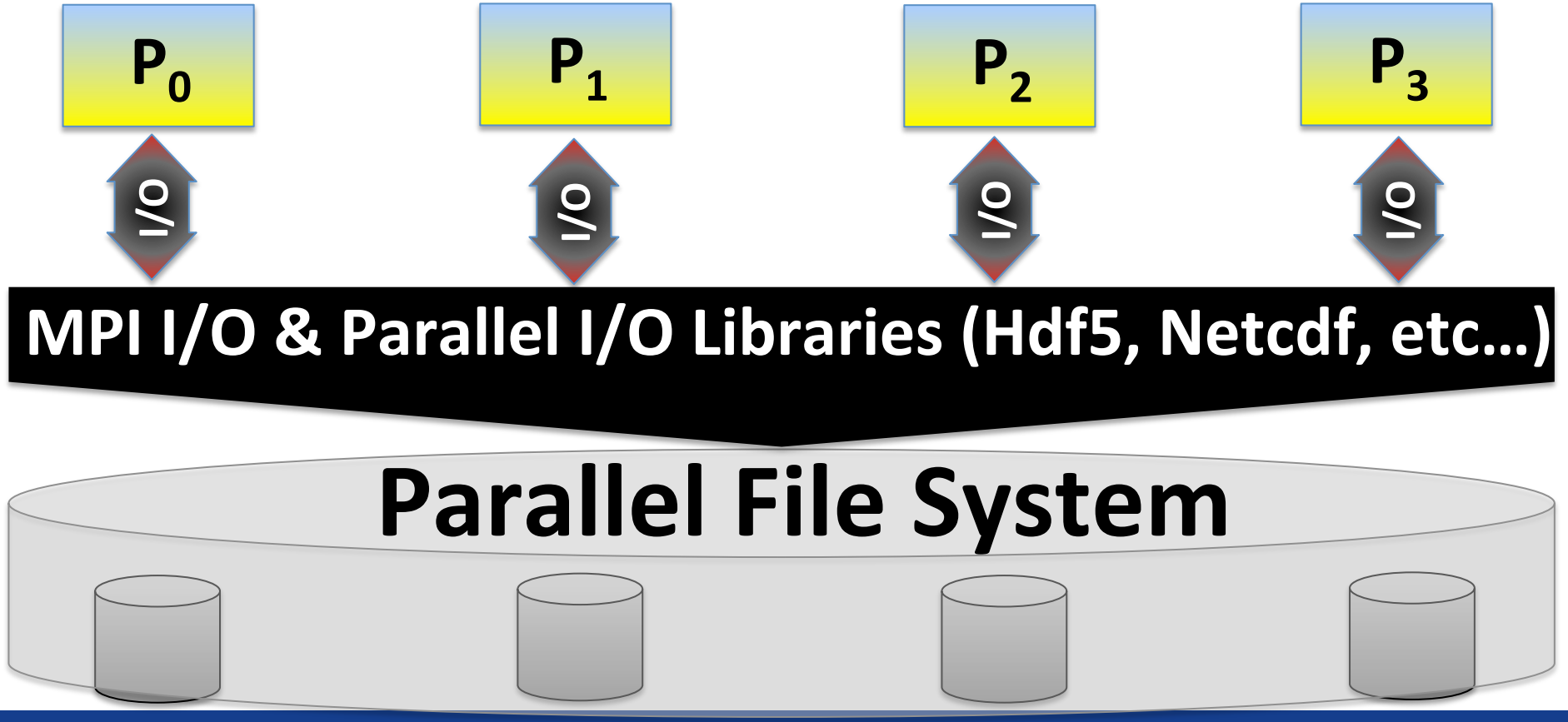
# Parallel I/O



# 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

# 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)

# Programming Parallel Paradigms

- Are the tools we use to express the parallelism for on a given architecture (see also SPMD, SIMD, etc... )
- They differ in how programmers can manage and define key features like:
  - parallel regions
  - concurrency
  - process communication
  - synchronism




# Fundamental Tools of Parallel Programming

# Phases of an MPI Program

1. Startup
   - Parse arguments (mpirun may add some!)
   - Identify parallel environment and rank of process
   - Read and distribute all data
2. Execution
   - Proceed to subroutine with parallel work (can be same of different for all parallel tasks)
3. Cleanup

CAUTION: this sequence may be run only once

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_FINALIZE(ierr)

end program bcast
```

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD
```

**$P_0$**

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

**$P_1$**

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

**$P_2$**

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

**$P_3$**

myrank = ??
ncpus = ??
imesg = ??
ierr = ??
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
```

**$P_0$**

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = ??
ncpus = ??
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)

  call MPI_COMM_SIZE(comm, ncpus, ierr)

  call MPI_COMM_RANK(comm, myrank, ierr)
```

**$P_0$**

myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = ??
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)

  call MPI_COMM_SIZE(comm, ncpus, ierr)

  call MPI_COMM_RANK(comm, myrank, ierr)
```

**$P_0$**

myrank = 0
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = ??
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
       " and my message content is ", imesg
```

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = 1
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 2
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 3
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
       " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)
```

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = 1
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 2
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 3
ierr = MPI_SUC...
comm = MPI_C...

call MPI_BCAST( imesg, 1, MPI_INTEGER, 0, comm, ierr )

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = 1
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 2
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 3
ierr = MPI_SUC...
comm = MPI_C...

call MPI_BCAST( imesg, 1, MPI_INTEGER, 0, comm, ierr )

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg
```

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_FINALIZE(ierr)
```

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

```
program bcast

  implicit none

  include "mpif.h"

  integer :: myrank, ncpus, imesg, ierr
  integer, parameter :: comm = MPI_COMM_WORLD

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(comm, myrank, ierr)
  call MPI_COMM_SIZE(comm, ncpus, ierr)

  imesg = myrank
  print *, "Before Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_BCAST(imesg, 1, MPI_INTEGER, 0, comm, ierr)

  print *, "After Bcast operation I'm ", myrank, &
      " and my message content is ", imesg

  call MPI_FINALIZE(ierr)

end program bcast
```

**$P_0$**

myrank = 0
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_1$**

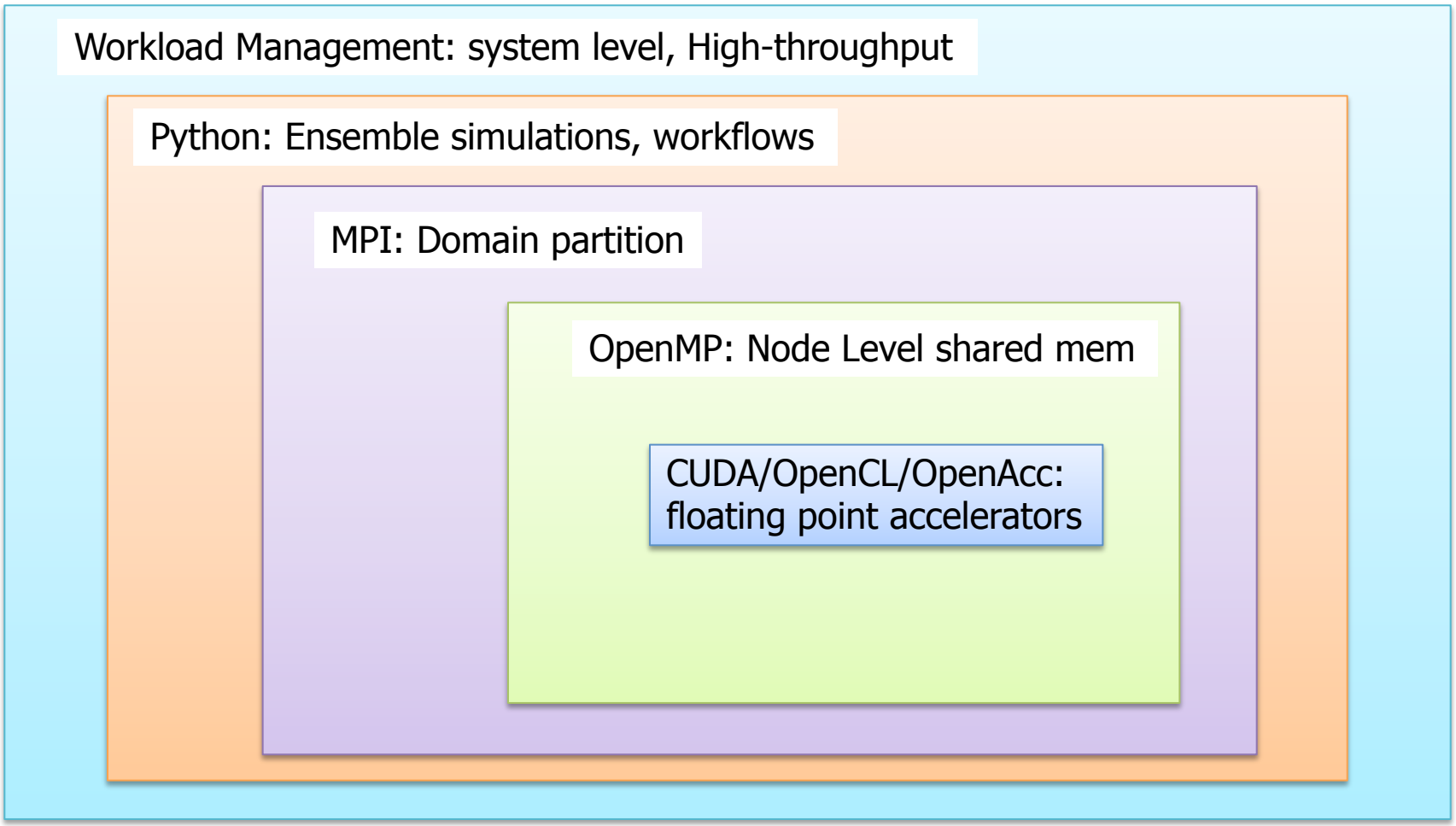
myrank = 1
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...

**$P_2$**

myrank = 2
ncpus = 4
imesg = 0
ierr = MPI_SUCC
comm = MPI_C...

**$P_3$**

myrank = 3
ncpus = 4
imesg = 0
ierr = MPI_SUC...
comm = MPI_C...


Workload Management: system level, High-throughput
Python: Ensemble simulations, workflows
MPI: Domain partition
OpenMP: Node Level shared mem
CUDA/OpenCL/OpenAcc:
floating point accelerators

# Thanks for your attention!!