Code Parallelization

a guided walk-through

m.cestari@cineca.it

f.salvadore@cineca.it

Summer School ed. 2015

24th Summer School on PARALLEL COMPUTING

Code Parallelization

two **stages** to write a parallel code

- **problem domain**
  - algorithm
- **program domain**
  - implementation

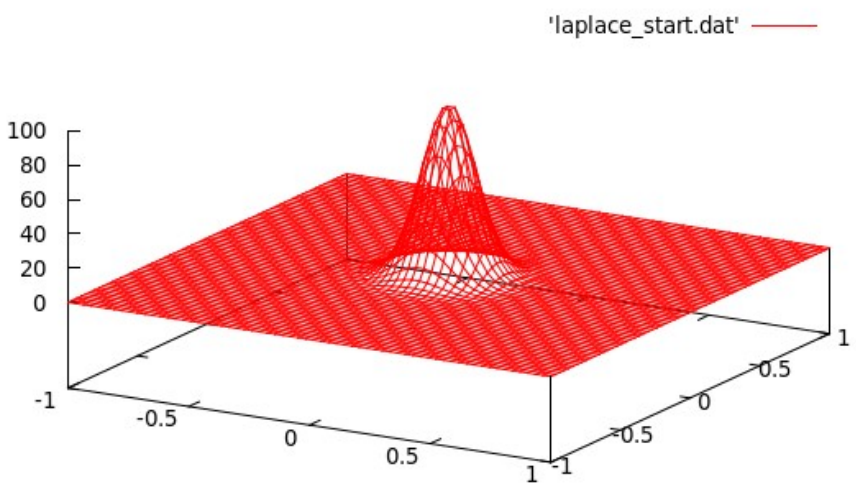
Code Parallelization

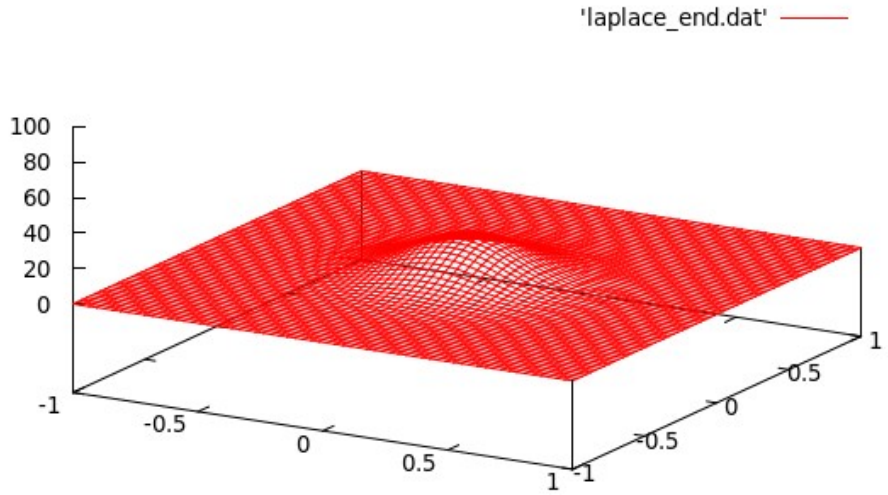
two **stages** to write a parallel code

- **problem domain**
  - algorithm
- **program domain**
  - implementation

Problem domain

- Naive iterative solver of Laplace equation for a variable T
  - Start with a Gaussian field
  - Iterate replacing each value with the mean value of the four neighboring points
  - Stop when either the maximum amount of iterations or the convergence is reached





Problem domain

- Analyze the algorithm. In principle (let us skip for the Laplace example):
  - Is the serial algorithm suitable for a a distribute parallel MPI implementation?
  - Is the serial algorithm still the best wrt performances for an MPI version of the code?

- Identify the most **computationally demanding** parts of the problem
  - But remember that an MPI parallelization is difficult to develop incrementally

Concurrency

Find concurrency:

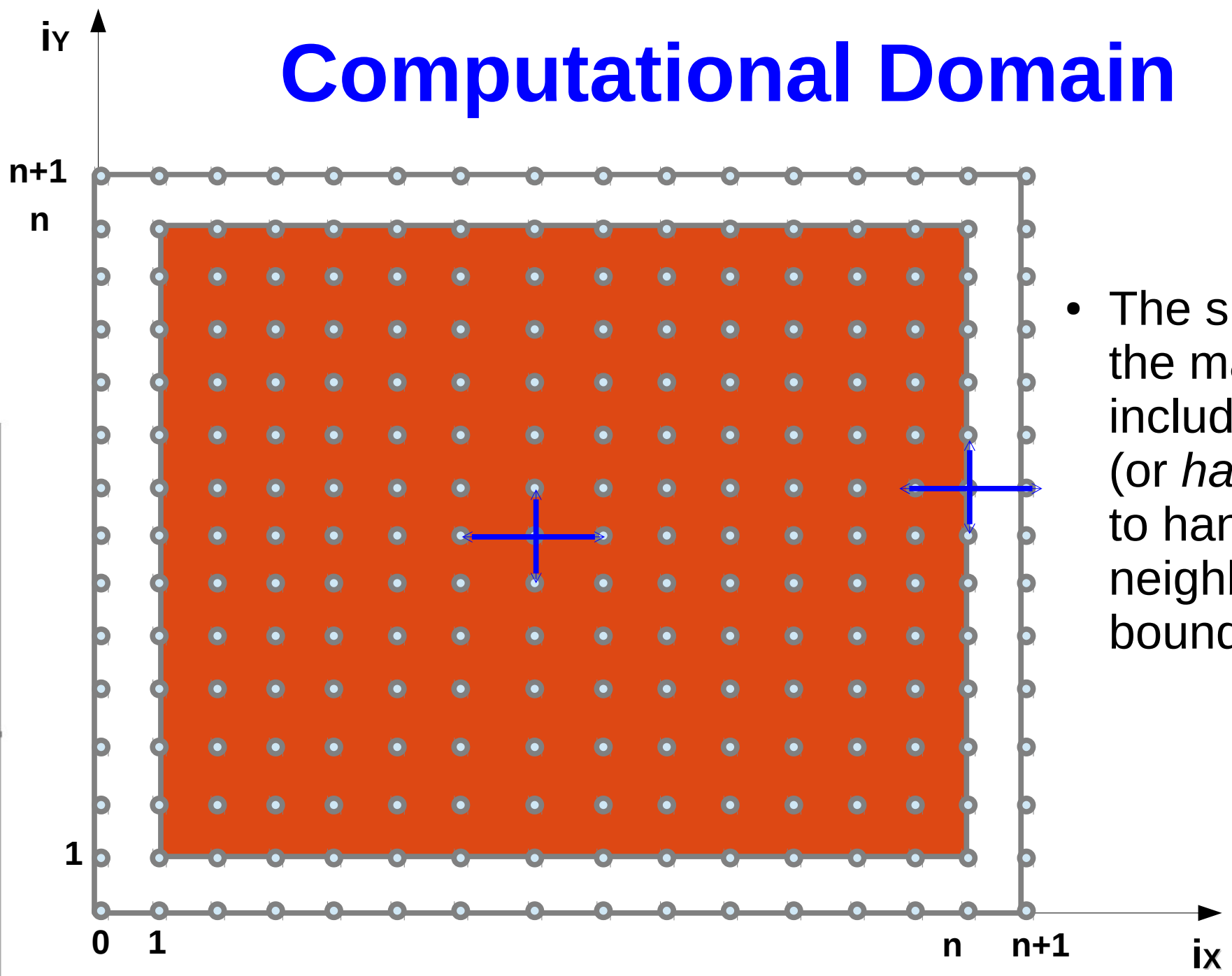
- **similar** operations that can be applied to **different parts** of the data structure
- domain **decomposition**: divide data into chunks that can be operated concurrently
  - a task works only **its chunk** of data
  - map **local** to **global** variables

Dependencies

Handle dependencies among tasks:

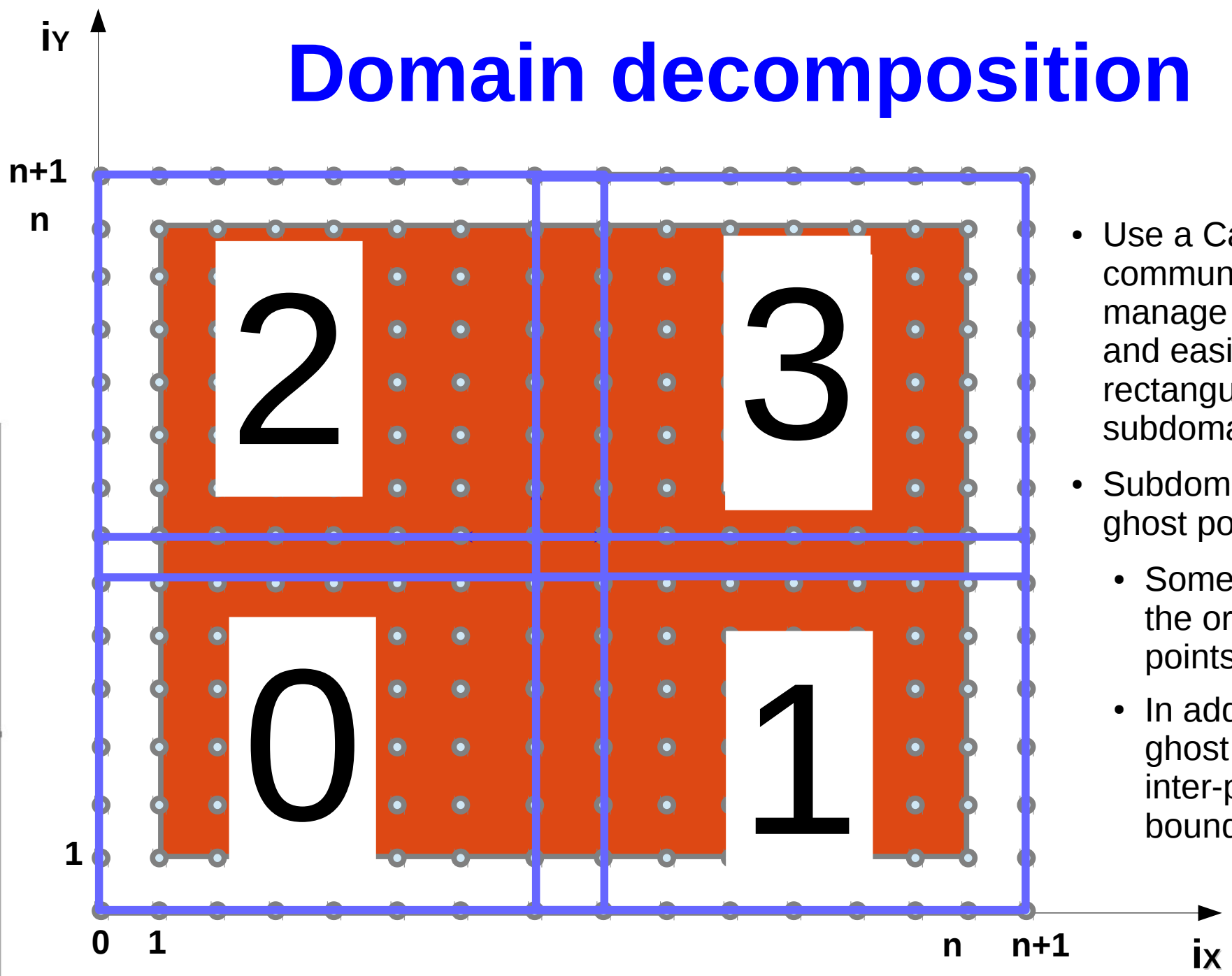
- Tasks needs access to some portion of another task local data (**data sharing**)
- Understand the kind and the amount of communications among processes required to make anything consistent

Computational Domain



- The shape of the matrixes include ghost (or *halo*) points to handle (the neighbour of) boundary points

Domain decomposition



- Use a Cartesian communicator to manage the processes and easily map them to rectangular subdomains
- Subdomains need ghost points too
  - Some of them are the original ghost points
  - In addition there are ghost points among inter-process boundaries

Program domain

2 different **stages** to parallelize a serial code

- **problem domain**
  - algorithm
- **program domain**
  - Implementation **(the fun part)**

The serial code: Laplace equation

```
program laplace
  [ … variable declarations … ]
  [ … input parameters ... ]
  [ … allocate variables … ]
  [ … initialize field … ]
  [ … print initial output … ]

  [ … computational core … ]

  [ … print final output … ]
  [ … deallocate variables … ]
end program laplace
```

```
do while (var > tol .and. iter <= maxIter)
    iter = iter + 1
    var = 0.d0
    do j = 1, n
      do i = 1, n
        Tnew(i,j) = 0.25d0 * (T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
        var = max(var, abs( Tnew(i,j) - T(i,j) ))
      end do
    end do

    Tmp =>T; T =>Tnew; Tnew => Tmp;

    if( mod(iter,100) == 0 ) &
      write(*,"(a,i8,e12.4)") ' iter, variation:', iter, var

  end do
```

The exercises

- (1) Develop an MPI parallel version of the laplace.f90/laplace.c serial codes (init and save functions are in init_save.f90/c files)
  - (a) Start with a MPI blocking implementation
  - (b) Try to enhance the solution using MPI non blocking calls
- (2) Add the OMP parallelization to the blocking MPI version to finally develop an hybrid MPI-OMP implementation of the code
  - Explore the different thread support levels

1.(a) Hints

- First create the Cartesian communicator
  - And find the ranks of the neighboring processes
- Define the sizes of the domain for each rank
  - Also define the offsets of the sub-domains with respect to the global domain
  - If possible try to handle the remainders, otherwise force a constraint
- After that, **init_field** is easy to parallelize: **ind2pos** (the function which maps the index to the position in the grid) remains unchanged provided that the global indexes are passed to it
- The print function (**save_gnuplot**) parallelization can be postponed: use the error at each time step to know if the results are correct
  - To parallelize it, let the rank=0 collect all the fields (just for didactic purposes, MPI I/O is the right way)
- At each iteration update the ghost points with the boundary points of the neighboring processes
  - MPI_Sendrecv may be a good choice
  - Declare, allocate and use buffers to perform the communications

1.(b) Hints

- In spite of MPI_Sendrecv non blocking MPI calls can be employed
  - MPI_Isend, MPI_Irecv, …
- But, how to make them useful to enhance the scalability?
  - Since the MPI communications are needed only for ghost nodes some operations can be performed simultaneously
  - Which operations? The operations which do not involve the ghost points...
- As always, man is your friend:

  man MPI_Init

2. Hints

- To mix MPI and OpenMP the simplest way is to open the OMP parallel region just around the computational core (the iteration loop)
  - MPI_THREAD_SINGLE (i.e., MPI_Init) version
- But the parallel region may be enlarged to include the MPI communications
  - If the communications are performed by the master thread MPI_THREAD_FUNNELED is enough
  - The communications may overlap with the computations if a technique like the MPI non blocking one is adopted
  - What about OMP schedule?
- The parallel region may be enlarged more including the entire while loop
  - Now MPI_THREAD_SINGLE could be employed to overlap pointer exchange and the MPI reduction for the error
  - Beware of the OMP barriers!
- And what about having different threads performing the different communications?
  - MPI_THREAD_MULTIPLE is needed
- Beware: use (and check) an MPI implementation supporting threads, e.g.
  - module load profile/advanced autoload intelmpi/5.0.1—binary
  - to compile, activate the thread enabled MPI library: mpif90 -mt_mpi

Misura delle performance / 1

- Strong scaling - Griglia 5000 $\times$ 5000 - 200 iterate
- Configuriamo l'ambiente e completiamo le tabelle
  - `module load module load profile/advanced`
  - `module load intel/cs-xe-2015--binary intelmpi/5.0.1--binary`
- MPI blocking *vs non-blocking*

MPI	*1*	*2*	*4*	*8*	*16*	*32*
Blocking						
Non-blocking						

- *MPI+OMP (MPI_THREAD_SINGLE version)*

MPI/OMP	*1*	*2*	*4*	*8*	*16*
1					
2					
4					
8					
16					
32					

- *Attenzione al settaggio dei processi per il caso ibrido e multi-nodo!*

Misura delle performance / 2

- Weak scaling - Griglia 800 $\times$ 800 per processo/thread - 200 iterate
- MPI blocking *vs non-blocking*

MPI	*1*	*2*	*4*	*8*	*16*	*32*
Blocking						
Non-blocking						

- *MPI+OMP (MPI_THREAD_SINGLE version)*

MPI/OMP	*1*	*2*	*4*	*8*	*16*
1					
2					
4					
8					
16					
32					

- *Conviene l'ibrido?*

More hints... / 1

- Initialize MPI:
  - MPI_Init / MPI_Comm_rank / MPI_Comm_size
- Input
  - Make only rank=0 read from input
  - MPI_Bcast the 3 input numbers to all the processes
- Cartesian topology for processes
  - MPI_Dims_create – decompose the number of processes in a rectangular way cart_dims(:)
  - MPI_Cart_create – create the Cartesian communicator
  - MPI_Cart_coords – find the coordinates of my process cart_coord(:)
  - MPI_Cart_shift (in x and y) – find the ranks of neighboring processes
- Associate the cartesian topology to the computational grid
  - Find for each process the sub-domain size and the start indexes wrt to the global domain (in x and y): mysize_y, mysize_y, mystart_x, mystart_y
    - mysize_x = n/cart_dims(1)
    - mystart_x = mysize_x *cart_coord(1)
  - Handle the remainders or force to be multiple (...)
- Allocate T, Tnew, and the buffers (4 send and 4 receive buffers), including the ghost points (size=mysize_x+2)

More hints... / 2

- Parallelize init_fields
  - Pass mystart_x,mystart_y,mysize_x,mysize_y as arguments
  - Modify the loop bounds from 0 to mysize_x/y+1
  - Modify the call to ind2pos (pass ix+mystart_x instead of ix)
- While loop:
  - Modify the loops bounds (from 1 to mysize_x/y)
  - MPI_Allreduce to the error variable (max among all the processes)
  - You are ready to check the results, just print the error variable after one step: serial and parallel codes must give the same results

More hints... / 3

- Communications
  - Just before the main update loop
  - 4 MPI_Sendrecv are enough: send to left + recv from right, send to right + recv from left, send to top + recv from bottom, send to bottom + recv from top
- Send to left + recv from right
  - Copy left boundary to a buffer
  - buffer_s_rl(1:mymsize_y) = T(1,1:mymsize_y)
  - Send to left and receive from right
  - MPI_Sendrecv(buffer_s_rl, mymsize_y, MPI_DOUBLE_PRECISION, dest_rl, tag,
    buffer_r_rl, mymsize_y, MPI_DOUBLE_PRECISION, source_rl, tag,
    cartesianComm, status, ierr)
  - Copy the received buffer
  - if(source_rl >= 0) T(mymsize_x+1,1:mymsize_y) = buffer_r_rl(1:mymsize_y)
  - Why is the if required? Because MPI_Cart_shift return MPI_PROC_NULL when a neighboring process does not exist
  - MPI_Sendrecv can correctly handle it (no send or receive is performed in that case)
  - But the copy back from buffer to T must be avoided (otherwise T would be filled with unexpected values)