



# ***Code Parallelization***

## ***a guided walk-through***

f.salvadore@cineca.it

2016



# Code Parallelization

two **stages** to write a parallel code

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



# Code Parallelization

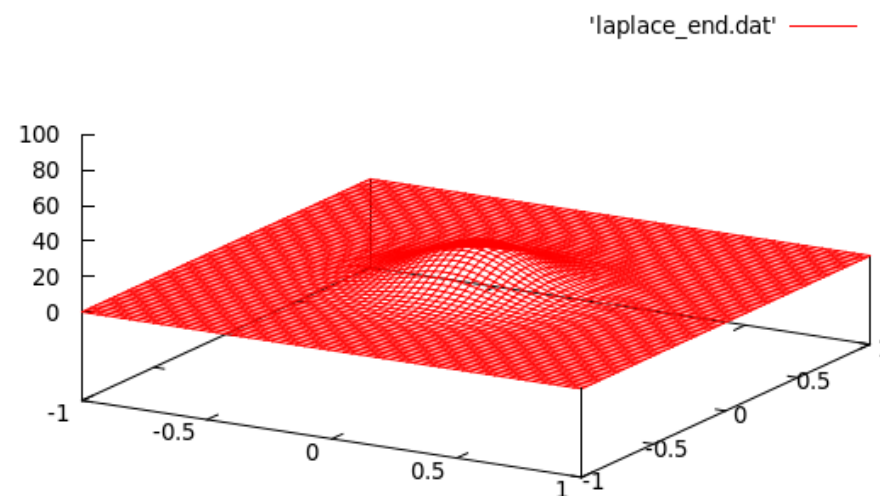
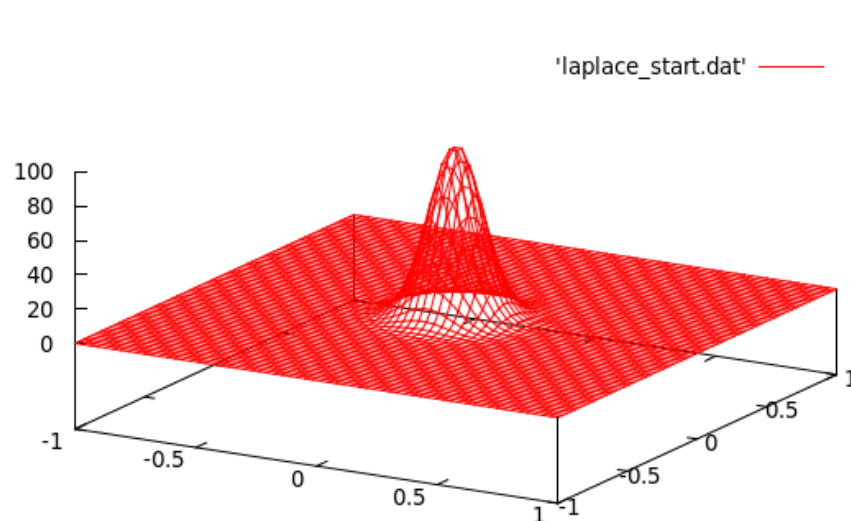
two **stages** to write a parallel code

- **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 (trivial for the Laplace example):
  - Is the serial algorithm suitable for a distributed 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

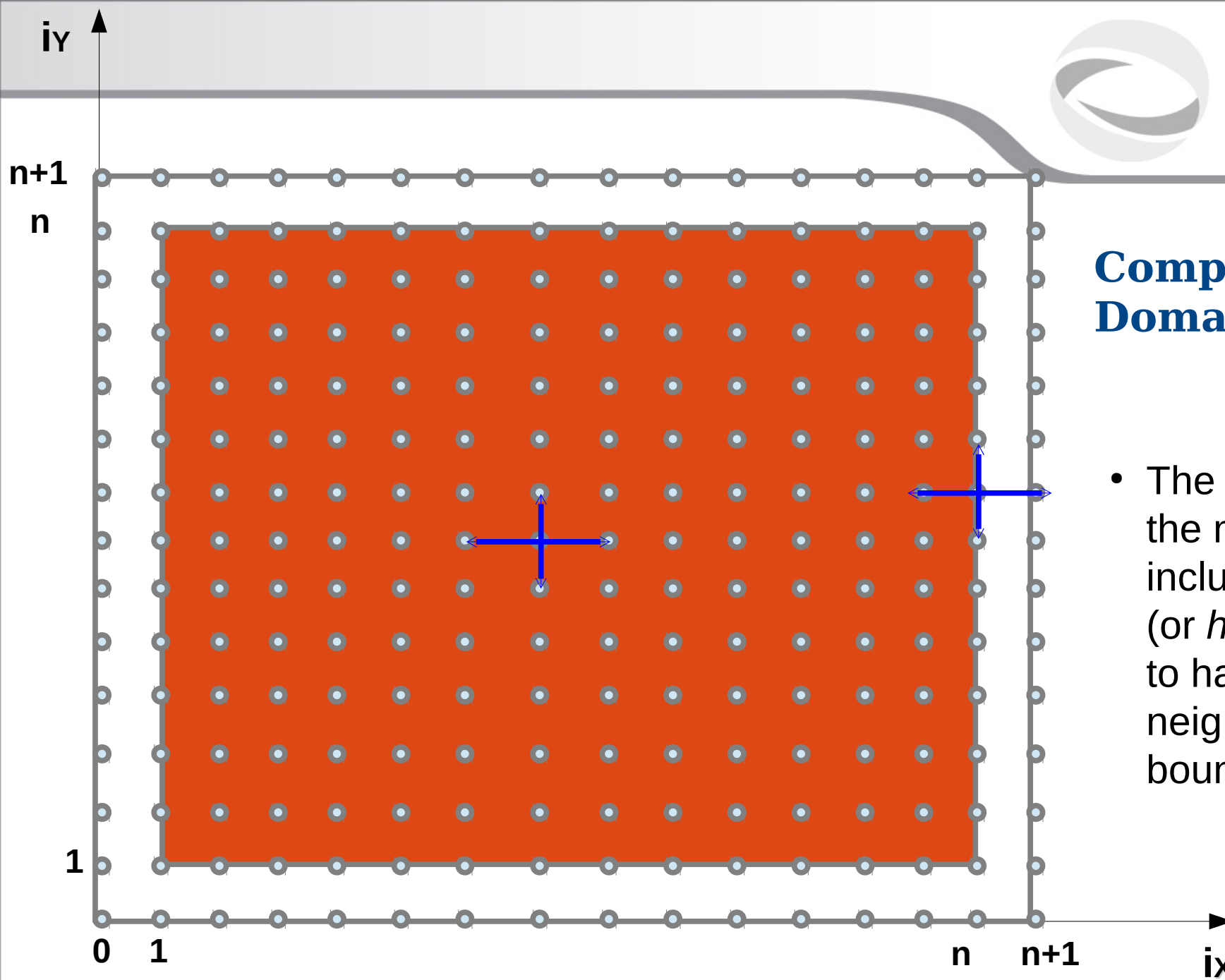
Handle dependencies among tasks:

- 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

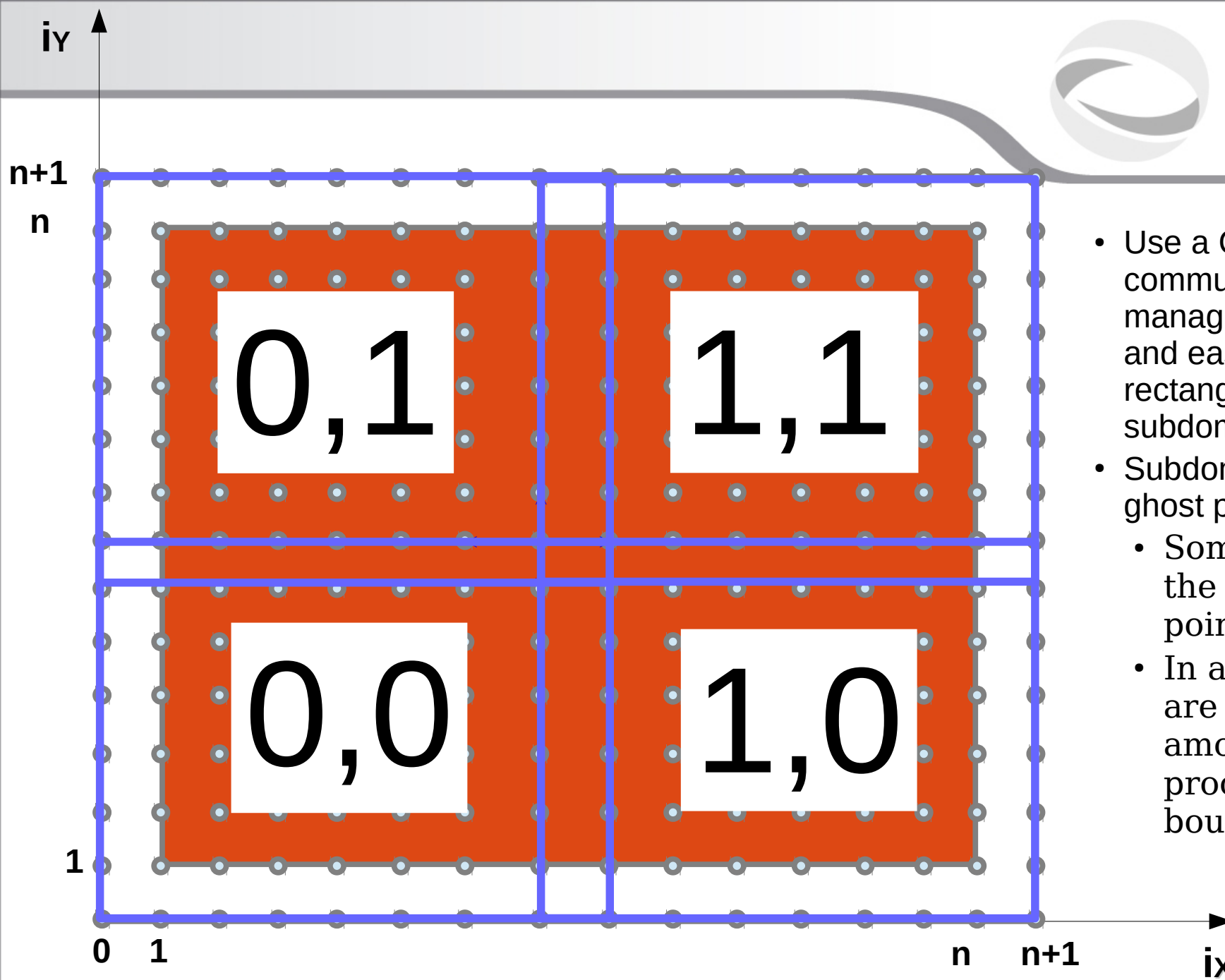


## Computational Domain

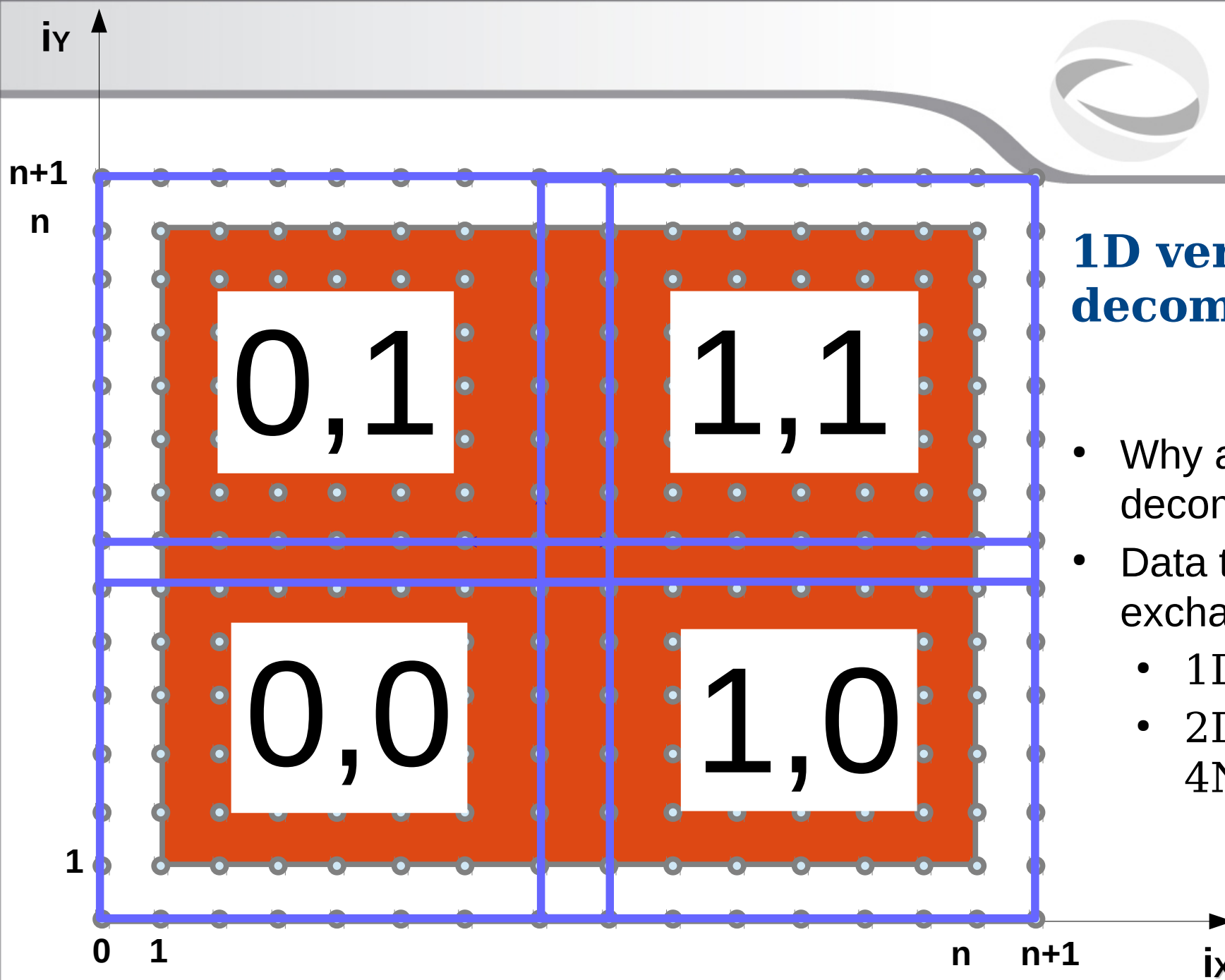
- The shape of the matrixes include ghost (or *halo*) points to handle (the neighbour of) boundary points







- 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



## 1D versus 2D decomposition

- Why a 2D decomposition?
- Data to be exchanged:
  - 1D:  $2N$
  - 2D:  $4N/\sqrt{N_{\text{proc}}}$



# Program domain

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

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



# 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 tasks

- (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 basic MPI implementation using a Cartesian topology and blocking communications
  - (b) Try to enhance the solution using advanced features
- (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



# MPI Basic - Hints / 1

- 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
  - might be postponed: check the error at each time step to know if the results are correct
  - to parallelize it, let the rank=0 collect all the fields and print (just for didactic purposes) but the right way is using MPI I/O
- 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



# MPI Basic - Hints / 2

- 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\_Comm\_rank on 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_x`, `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`). Ghosts not needed for buffers.
- Declare everything you need!





## MPI Basic - Hints / 3

- **Parallelize `init_fields`**
  - Pass `mystart_x, mystart_y, mymsize_x, mymsize_y` as arguments
  - Modify the loop bounds from 0 to `mymsize_x/y+1`
  - Modify the call to `ind2pos` (pass `ix+mystart_x` instead of `ix`)
- **Parallelize print function (`save_gnuplot`) parallelization**
  - to parallelize it, let the `rank=0` collect all the fields and print ASCII (just for didactic purposes)
  - the right way would be MPI I/O
- **To focus on MPI advanced features, parallel versions of `init_fields` and `save_gnuplot` are already provided**



## MPI Basic - Hints / 3

- Parallelize **init\_fields**
  - Pass `mystart_x, mystart_y, mymsize_x, mymsize_y` as arguments
  - Modify the loop bounds from 0 to `mymsize_x/y+1`
  - Modify the call to `ind2pos` (pass `ix+mystart_x` instead of `ix`)
- Parallelize print function (**save\_gnuplot**) parallelization
  - to parallelize it, let the `rank=0` collect all the fields and print ASCII (just for didactic purposes)
  - the right way would be MPI I/O
- **To focus on MPI advanced features, parallel versions of `init_fields` and `save_gnuplot` are already provided**



## MPI Basic - Hints / 4

- Main compute loop:
  - Modify the loops bounds (from 1 to mymsize\_x/y)
  - MPI\_Allreduce to the error variable (max among all the processes)
  - You are ready to check the first results, just print the error variable after one step: serial and parallel codes must give the same results
- **To focus on MPI advanced features, the parallel versions (except for communications) are already provided**
- Now you can start adding the communications
  - Inside the main compute loop
  - Usually just before the updates



# MPI Basic - Hints / 5

- Communications
  - 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
- E.g., send to left + recv from right
  - Copy left boundary to a buffer
  - Send to left and receive from right
- Copy back the received buffer
  - A conditional statement is required: where and why?



## MPI Basic - Hints / 6

- Now probably you will face problems
  - Errors when compiling: check the arguments of MPI calls, the MPI types, and for Fortran the kinds
  - Start verifying that the MPI code still works using 1 processor (mpirun -np 1 ...)
  - Then try to add the decomposition only on one dimension (mpirun -np 2 ...)
  - You can check the residuals or you can also check the field to understand the origin of the error
- **Do not discourage! Parallelizing a code –even simple – is not straightforward**



## MPI Advanced - Improvements

- So far we have a basic MPI parallelization of the original serial code
- Actually many improvements are possible
  - which may be possibly mixed
  - two common possibilities

Use **non-blocking Communications** and **overlap** them with computations

**Derived datatypes**  
Avoid copies on buffers even for not contiguous memory regions



## MPI Advanced - (1) Overlap communications with computations

- 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* (and the web, of course) is your friend:  
man MPI\_Init



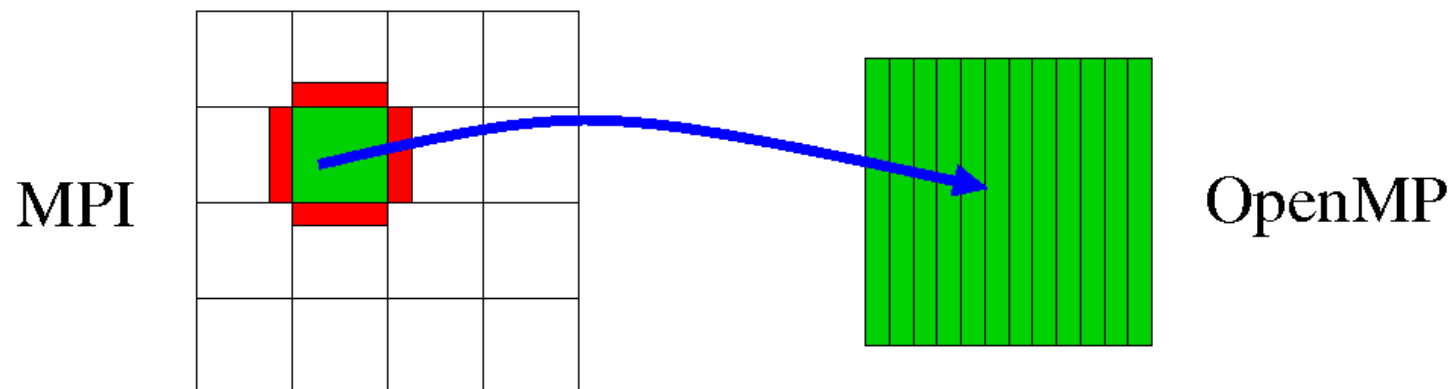
## (2) Using derived datatypes

- Restart from basic MPI version
- So far we have been using buffers as temporary storage for non-contiguous memory regions to send/recv (rows for Fortran and columns for C)
- But this can be avoided making the code more readable and possibly improving the performances
- Create two MPI derived datatypes (actually just one is really mandatory)
  - A type for a matrix row: which type is needed in Fortran? And in C?
  - A type for a matrix column: which type is needed in Fortran? And in C?
- Then send/recv only 1 element of this type
  - No buffer is needed!
  - Just pass to MPI\_Sendrecv the first element of the submatrix to pass and specify one element of the derived types to pass
  - Hint: do not forget to commit the type after creation!



## (1) MPI + OpenMP - Hints

- To mix MPI and OpenMP the simplest way is to open the OMP parallel region just around the main computing loop (the update iteration loop from  $T$  to  $T_{new}$ )
  - No direct interaction between MPI and OpenMP
  - But `MPI_THREAD_FUNNELED` should be required according to the standard
  - Actually `MPI_THREAD_SINGLE` (i.e., `MPI_Init`) also usually works (at least for OpenMPI)
  - 5 minutes should be enough to complete the hybridization
- Remember to add the *openmp* compilation option





## (2) MPI + OpenMP - Hints

- But the parallel region may be enlarged to include the MPI communications
  - If the communications are performed by the master threads, `MPI_THREAD_FUNNELED` is enough
  - The communications may overlap with the computations: *master threads* performs the communications and then update the boundaries
  - At the same time, the other threads start doing bulk updating
  - Probably master threads collaborate after a while in doing that
  - The OMP schedule should be modified accordingly
- Remember
  - OMP master forces the code to be executed only by master thread
  - And the other threads go on



## (3) MPI + OpenMP - Hints

- The parallel region may be further enlarged including the entire while loop
  - `MPI_THREAD_SERIALIZED` must be employed
  - Now we can overlap pointer exchange and the MPI reduction for the error
- Some OMP barriers are needed: where and why?
- Use OMP single
  - to do tasks which must be executed only by one thread: e.g. `“iter=iter+1”`
  - Or for the `MPI_Allreduce`



## (4) MPI + OpenMP - Hints

- What about “each thread executing an MPI communication”?
  - You need `MPI_THREAD_MULTIPLE` support
  - Each thread performs a send/recv: how to implement in OpenMP?
  - The other threads immediately start the core updating loop...
  - Then wait for the other threads to finish (how?) and update the boundaries



# Evaluating performances

- The different versions can lead to different results in term of performances
  - But the actual improvements depend on several factors
  - And are probably limited for such a didactic example
  - Testing in **realistic scenarios** is mandatory
  - For our case let us consider a 5000x5000 grid

	1	2	...	256
MPI basic				
Overlap				
DDT				



## Evaluating performances / 2

- To evaluate the improvement given by the hybrid programming the scaling evaluation can be more complex
  - No improvement expected for such a simple case

$\frac{N\_MPI}{N\_OpenMP}$	1	2	...	256
1				
2				
4				
8				
16				