Foundations of High Performance Computing

Lecture 7: Domain Decomposition by means of MPI

2020-2021 Stefano Cozzini

"Foundation of HPC" course

DATA SCIENCE & SCIENTIFIC COMPUTING

Agenda

- Domain Decomposition approach
- A Real example

Domain Decomposition

- A fundamental technique in both parallel programming and MPI programming
- The way to solve very large problems in memory on distributed memory architecture
- Useful to know some trick of the trade to deal with that using MPI

What is domain decomposition?

- Perform a numerical simulation (e.g. solve an equation) on a domain of given shape
- Discretize the domain using a mesh or particles
- Split the work (and memory!) among N processors
- Can be done “geographically” or purely on the data, irrespective of their spatial location
- Domains can have complex shapes

Goals to achive..

- All processors having equal (in terms of wall- clock time) amounts of work

  ==> “Load balancing”

- Minimum number of communication steps between the processors

  ==> “Communication scheduling”

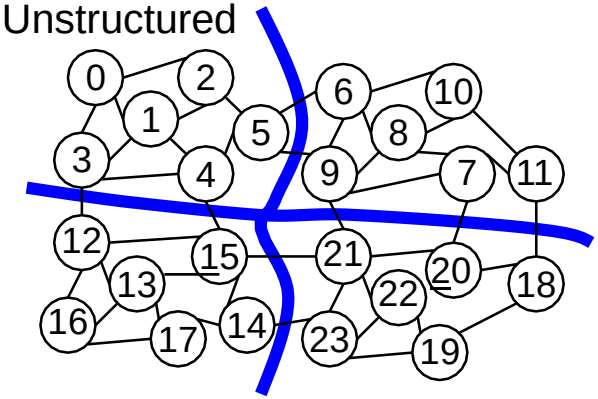
- Minimum amount of data that needs to be communicated. No deadlocks.

Domain Decomposition

Cartesian

0	1	2	6	7	8
3	4	5	9	10	11
12	13	14	18	19	20
15	16	17	21	22	23

Unstructured



Examples with 4 sub-domains

Non geographical approaches

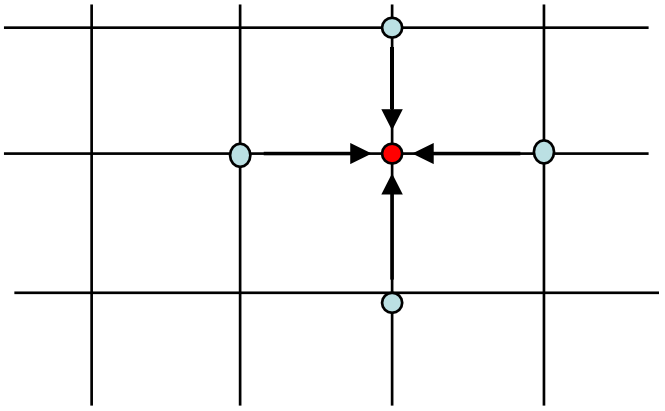
- Partition the data irrespective of their physical location in the computational domain
- Example: long range MD
  - having N interacting particles, each processor receives N/P particles. Since all have to communicate to all anyway, neighborhood relations are not important
  - Best load balance due to finest granularity
  - Easy to compute
  - No re-balancing needed if particles move

Cartesian coordinates
=
regular meshes

Operations on meshes

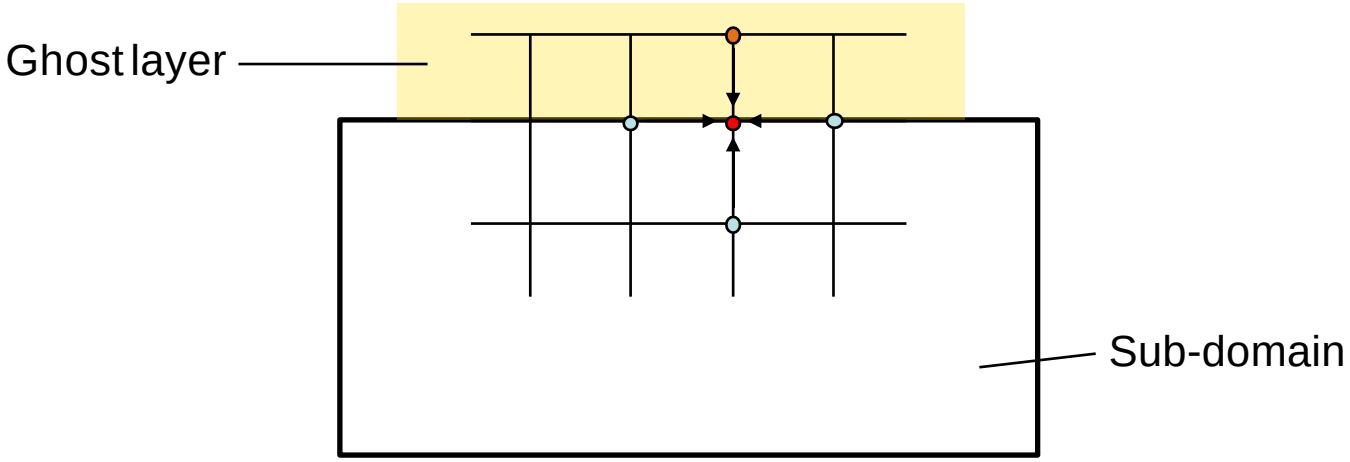
The values at mesh nodes are updated such as to numerically solve a given equation:

- **Finite differences** (Stencil methods)
  Values of mesh nodes are updated using the values of (any set of) nearby nodes.
- Ex: a 5 point stencil (diffusion equation)



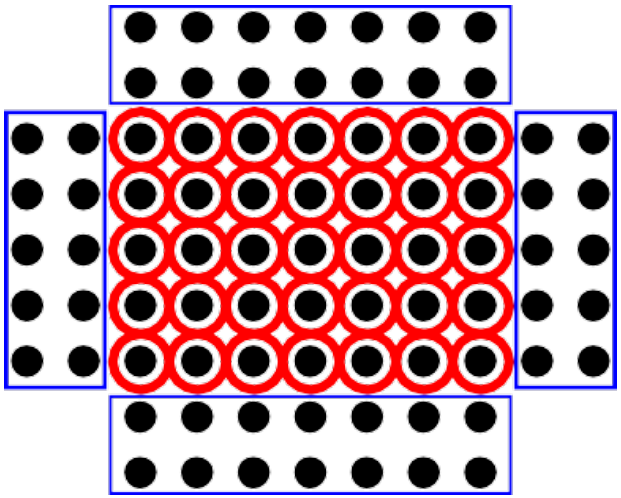
Operations on meshes

- Ghost layers around each sub-domain provide the needed values at the sub-domain boundaries



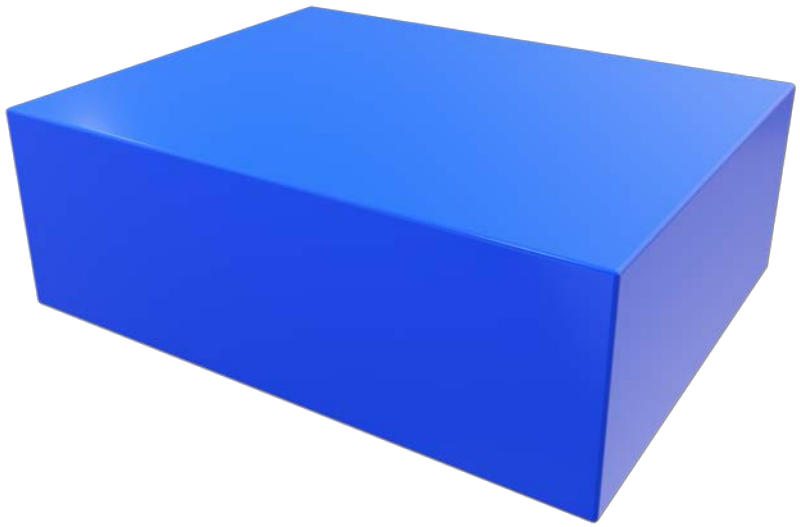
Ghosts cells or halo area

- To calculate the new grid points of a sub-domain, additional grid points from other sub-domains are needed.
- They are stored in halos (ghost cells, shadows)
- Halo depends on form of stencil



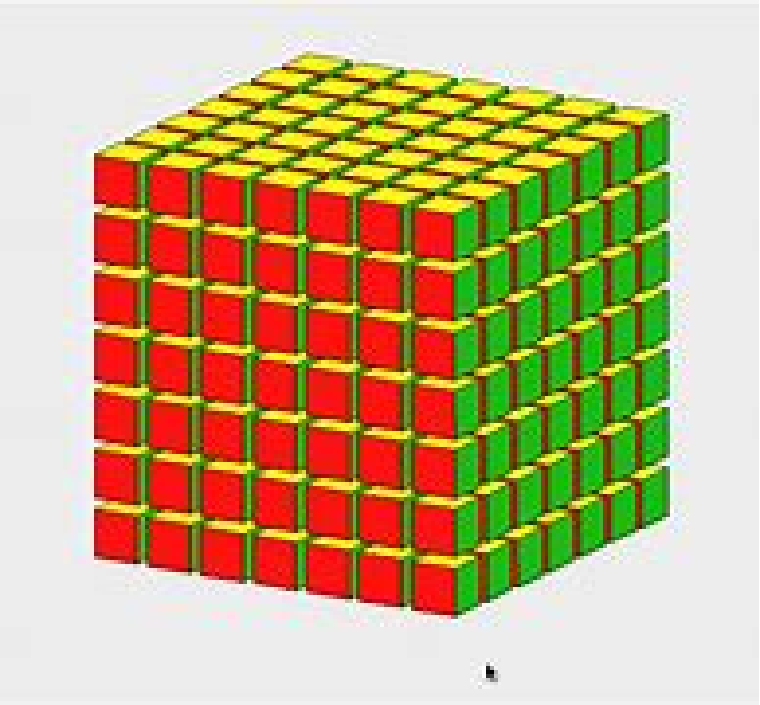
1. Splitting the domain..

- Consider a 3D box
- We can split along 1/2/3 dimensions
- Which is the best choice in term of performance (i.e scalability) ?



Example:a cube

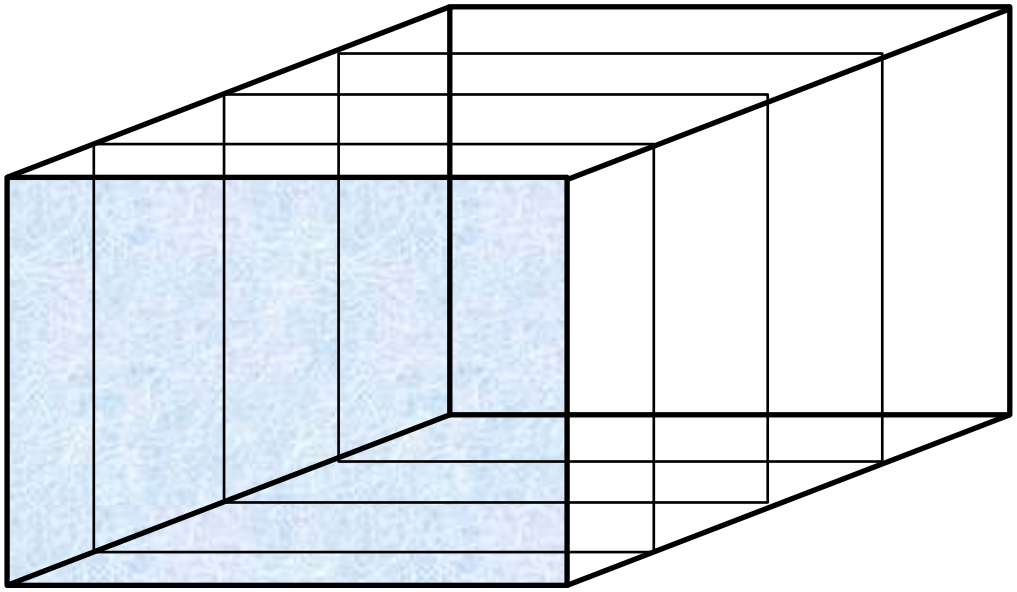
- Let us consider the following parameters:
  - N^3 size of the cubic matrix
  - P processor
  - Cyclic boundary condition:
    - two neighbours in each direction
  - W: width of the “ ghost cells” or halo
- Define in term of these parameters the cost of the communication message



Domain decomposition: 1D

- Splitting along 1 dimension:

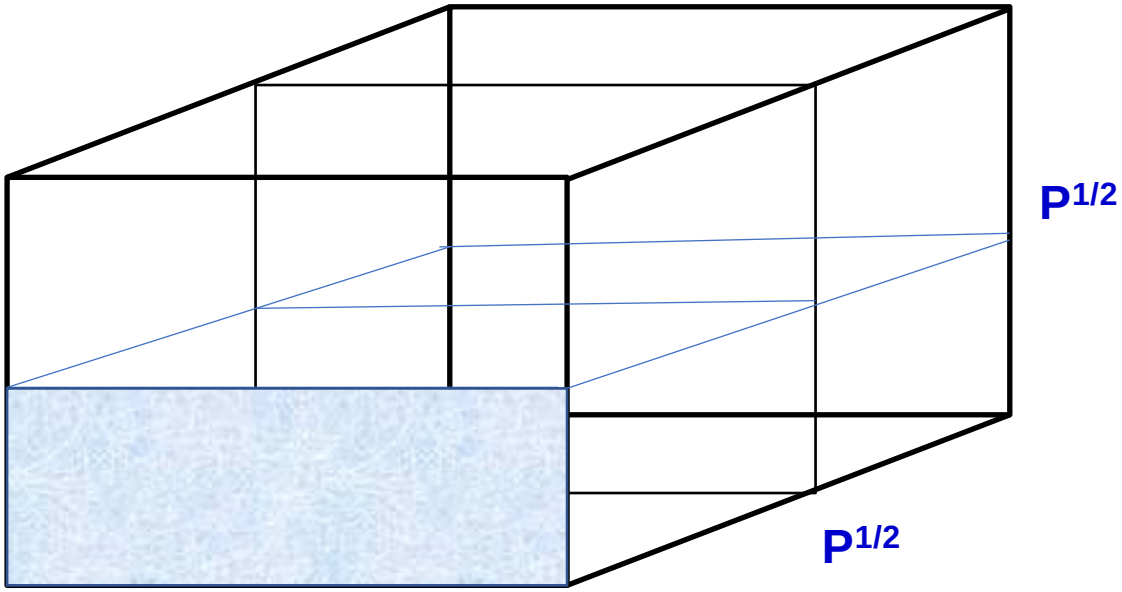
1D communication
$= n^2 * 2 * w * 1$



Domain decomposition: 2D

- Splitting along 2 dimension:

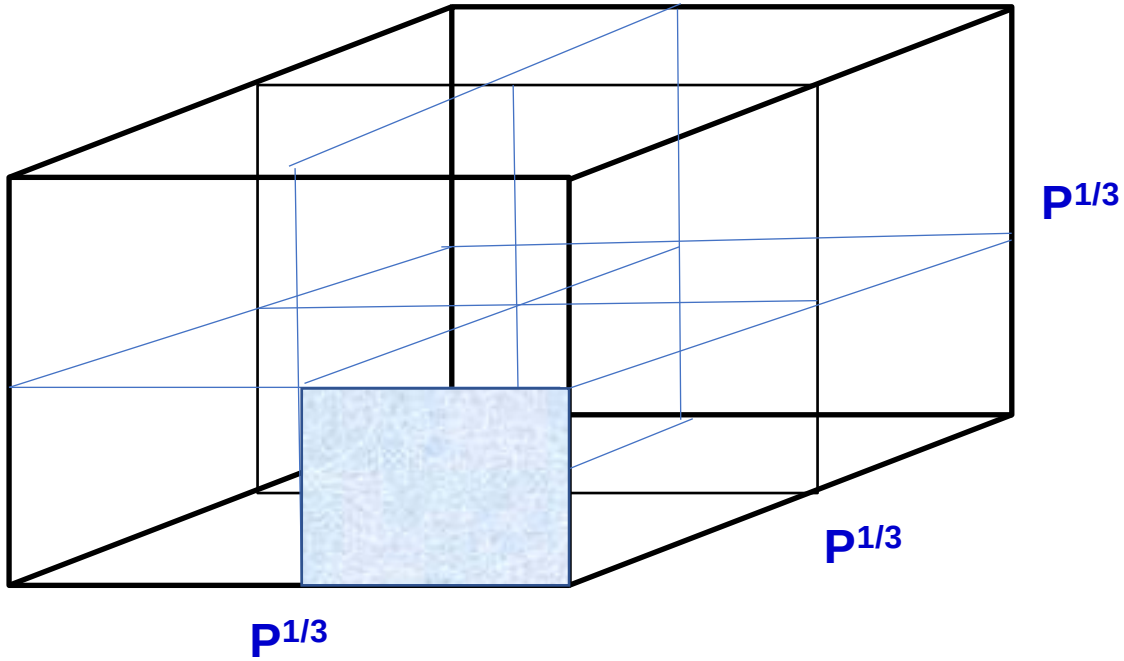
2D communication
$= n^2 * 2 * w * 2/P^{1/2}$



Domain decomposition: 3D

- Splitting along 3 dimension:

3D communication
$= n^2 * \mathbf{2} * w * \mathbf{3/P^{2/3}}$



What about scalability ?

Process	1D	2D	3D
2,00	2,00	2,83	3,78
4,00	2,00	2,00	2,38
8,00	2,00	1,41	1,50
16,00	2,00	1,00	0,94
32,00	2,00	0,71	0,60
64,00	2,00	0,50	0,38

Domain Decomposition 101

1. Split the domain into blocks.
2. Assign blocks to MPI-processes one-to-one.
3. Provide a "map" of neighbors to each process.
4. Write or modify your code so it only updates a single block.
5. Insert communication subroutine calls where needed.
6. Adjust the boundary conditions code.
7. Use "ghost cells/halo".

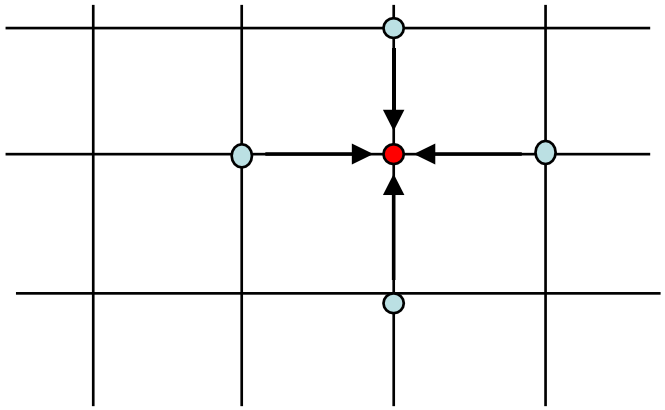
Real example: Jacobi solver..

- prototype for many stencil-based iterative methods in numerical analysis and simulation
- Basic form: solve the diffusion equation for a scalar function: $\Phi(\boldsymbol{r}, t)$

$$\frac{\partial \Phi}{\partial t} = \Delta \Phi \ ,$$

Straightforward 2D serial implementation: the stencil

```
do k = 1,kmax
 do i = 1,imax
  phi(i,k,t1) = 0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0)
              + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
  enddo
 enddo
```



All taken from reference 4

The full serial algorithm..

- Ensure that the code produces a converged result.

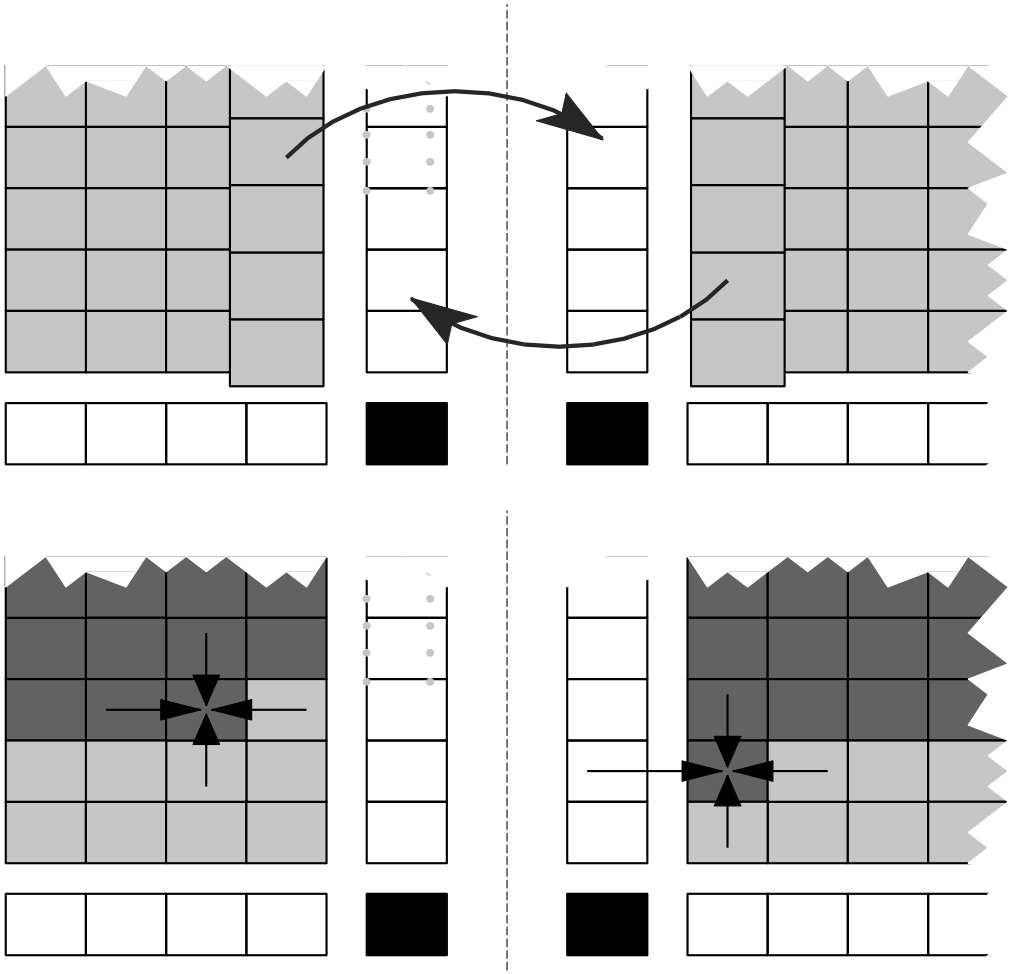
```
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
double precision :: maxdelta,eps
 integer :: t0,t1
 eps = 1.d-14 ! convergence threshold
 t0 = 0 ; t1 = 1
 maxdelta = 2.d0*eps
 do while(maxdelta.gt.eps)
 maxdelta = 0.d0
do k = 1,kmax
   do i = 1,imax
          phi(i,k,t1) = 0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0)
                      + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
          maxdelta = max(maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)))

  enddo
      enddo
! swap arrays
i = t0 ; t0=t1 ; t1=i
 enddo
```

What happens in the parallel implementation ?

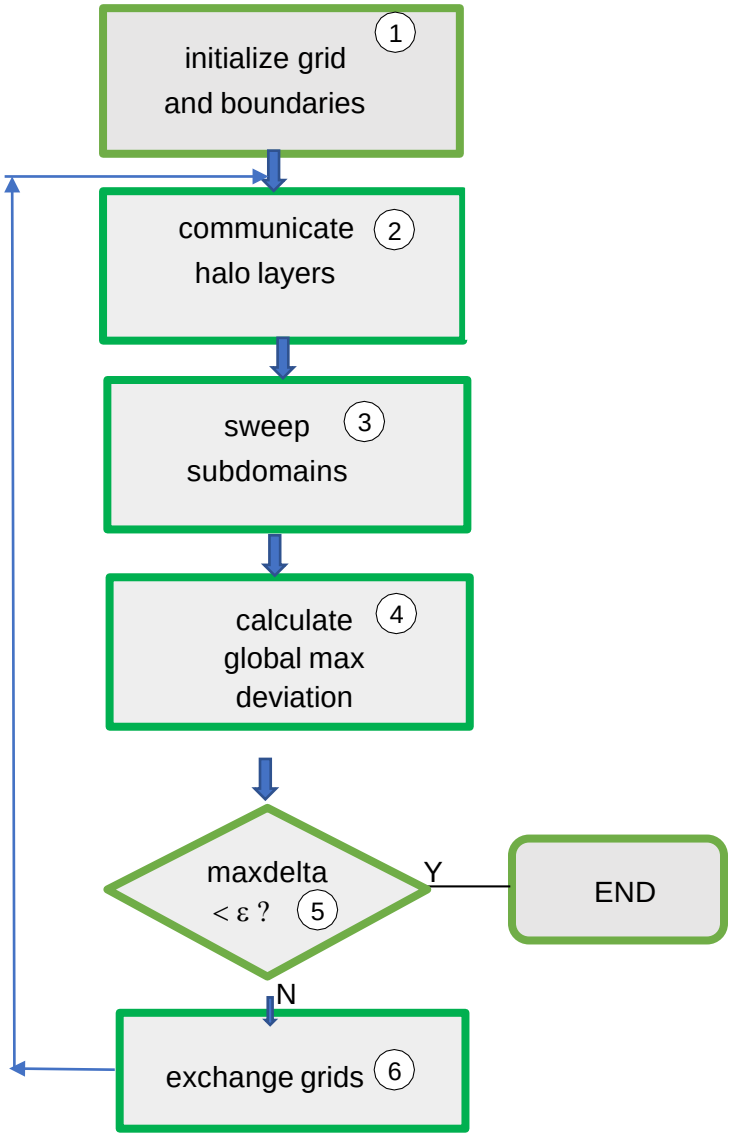
- The computational core works the same but:
  - Convergence criterion is no longer enough on subdomains but need to be computed globally

    ➔ requires a reduce operation among all processors
  - We need to take care of boundary conditions : Cell close to border require special care and require halo layers

The parallel data distribution..



The parallel algorithm



A few remarks

- Initialization is done with virtual topology

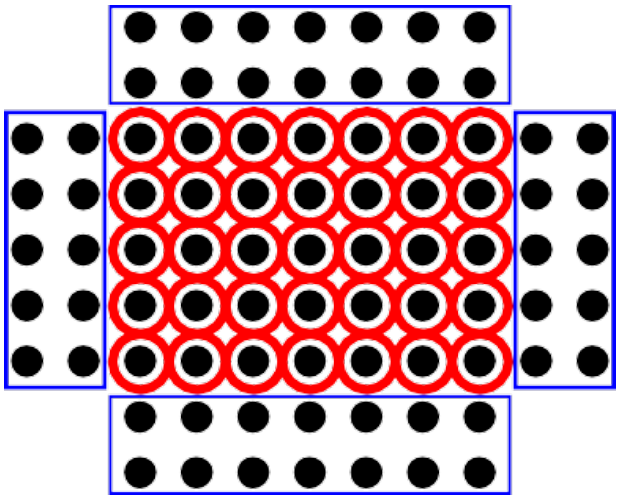
```
1  call MPI_Dims_create(numprocs, 3, proc_dim, ierr)
2
3  if(myid.eq.0) write(*,'(a,3(i3,x))') 'Grid: ', &
4      (proc_dim(i),i=1,3)
5
6  l_reorder = .true.
7  call MPI_Cart_create(MPI_COMM_WORLD, 3, proc_dim, pbc_check, &
8      l_reorder, GRID_COMM_WORLD, ierr)
9
10 if(GRID_COMM_WORLD .eq. MPI_COMM_NULL) goto 999
11
12 call MPI_Comm_rank(GRID_COMM_WORLD, myid_grid, ierr)
13 call MPI_Comm_size(GRID_COMM_WORLD, nump_grid, ierr)
```

Halo communication

- We use point to point communication to exchange halo layer.
- Point-to-point communication requires consecutive message buffers.
- Do we have contiguous location in memory for the halo ?

Not at all…

- only those boundary cells that are consecutive in the inner (i) dimension are also consecutive in memory (fortran column-major order..)
- Whole layers in the i-j, i-k, and j-k planes are never consecutive

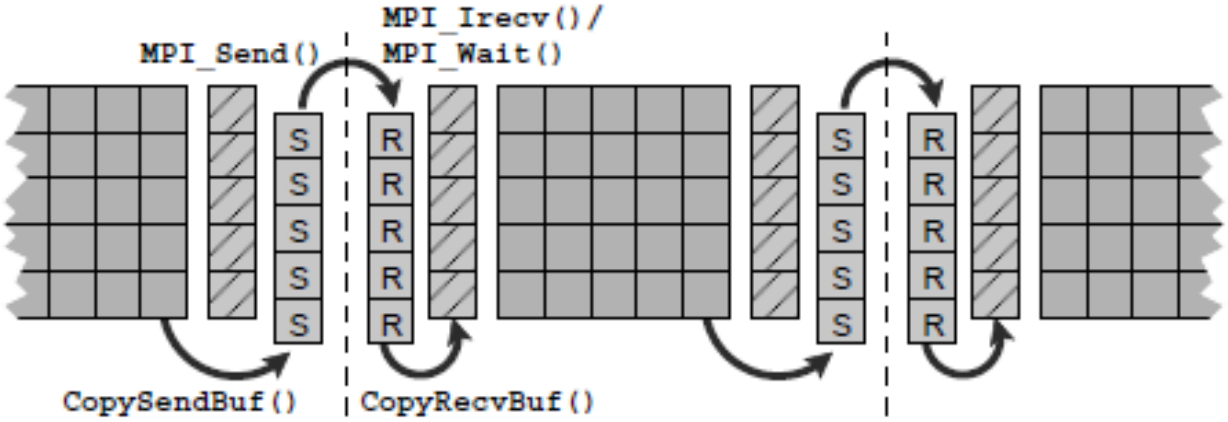


How to deal with this issue ?

- Send each chunk in a separate message discouraged:
  - This approach would flood the network with short messages, and latency has to be paid for every request
- BEST SOLUTION
  - intermediate buffer must be used to gather boundary data to be communicated to a neighbor’s ghost layer.

Halo exchange in one direction..

- We use two intermediate buffers per process, one for sending and one for receiving.



Dimension of the buffer…

- halo data can be different along different Cartesian directions

➔ the size of the intermediate buffer must be chosen to accommodate the largest possible halo

```
integer, dimension(1:3) :: totmsgsize

! j-k plane
totmsgsize(3) = loca_dim(1)*loca_dim(2)
MaxBufLen=max(MaxBufLen,totmsgsize(3))
! i-k plane
totmsgsize(2) = loca_dim(1)*loca_dim(3)
MaxBufLen=max(MaxBufLen,totmsgsize(2))
! i-j plane
totmsgsize(1) = loca_dim(2)*loca_dim(3)
MaxBufLen=max(MaxBufLen,totmsgsize(1))

allocate(fieldSend(1:MaxBufLen))
allocate(fieldRecv(1:MaxBufLen))
```

Halo exchange.

```
  do disp = -1, 1, 2
    do dir = 1, 3

      call MPI_Cart_shift(GRID_COMM_WORLD, (dir-1), &
                          disp, source, dest, ierr)

      if(source /= MPI_PROC_NULL) then
        call MPI_Irecv(fieldRecv(1), totmsgsize(dir), &
             MPI_DOUBLE_PRECISION, source, &
             tag, GRID_COMM_WORLD, req(1), ierr)
      endif    ! source exists

      if(dest /= MPI_PROC_NULL) then
        call CopySendBuf(phi(iStart, jStart, kStart, t0), &
                         iStart, iEnd, jStart, jEnd, kStart, kEnd, &
                         disp, dir, fieldSend, MaxBufLen)

        call MPI_Send(fieldSend(1), totmsgsize(dir), &
                      MPI_DOUBLE_PRECISION, dest, tag, &
                      GRID_COMM_WORLD, ierr)
      endif    ! destination exists

      if(source /= MPI_PROC_NULL) then
        call MPI_Wait(req, status, ierr)

        call CopyRecvBuf(phi(iStart, jStart, kStart, t0), &
                         iStart, iEnd, jStart, jEnd, kStart, kEnd, &
                         disp, dir, fieldRecv, MaxBufLen)
      endif    ! source exists

    enddo    ! dir
  enddo    ! disp
```

For full 3D implementation

- See github repo…

Exercises

- Familiarize with the program and start running it on different number of processors and different sizes.
- Implement in your preferred language a 2D Jacobi solver
- Next time we discuss performance model and scalability behaviour