23rd Summer School on PARALLEL COMPUTING

MPI Virtual topologies- Domain Decomposition case study

Andrew Emerson- a.emerson@cineca.it
SuperComputing Applications and Innovation Department

CINECA

contents

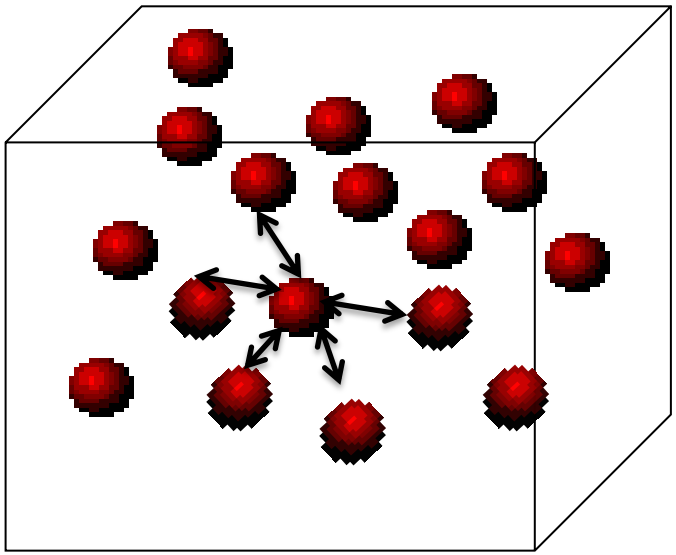
- Introduction
- Domain decomposition
- Examples
  - Game of Life
  - Classical molecular dynamics and scaling limits
- Final comments

introduction

- *Domain decomposition* refers to a general set of methods for solving a boundary value problem by splitting it into smaller boundary value problems.
- In mathematics often used to solve systems of partial differential equations.
- In computational chemistry and physics used to parallelise simulations over large number of interacting particles.

example

- Consider example of a system of interacting particles



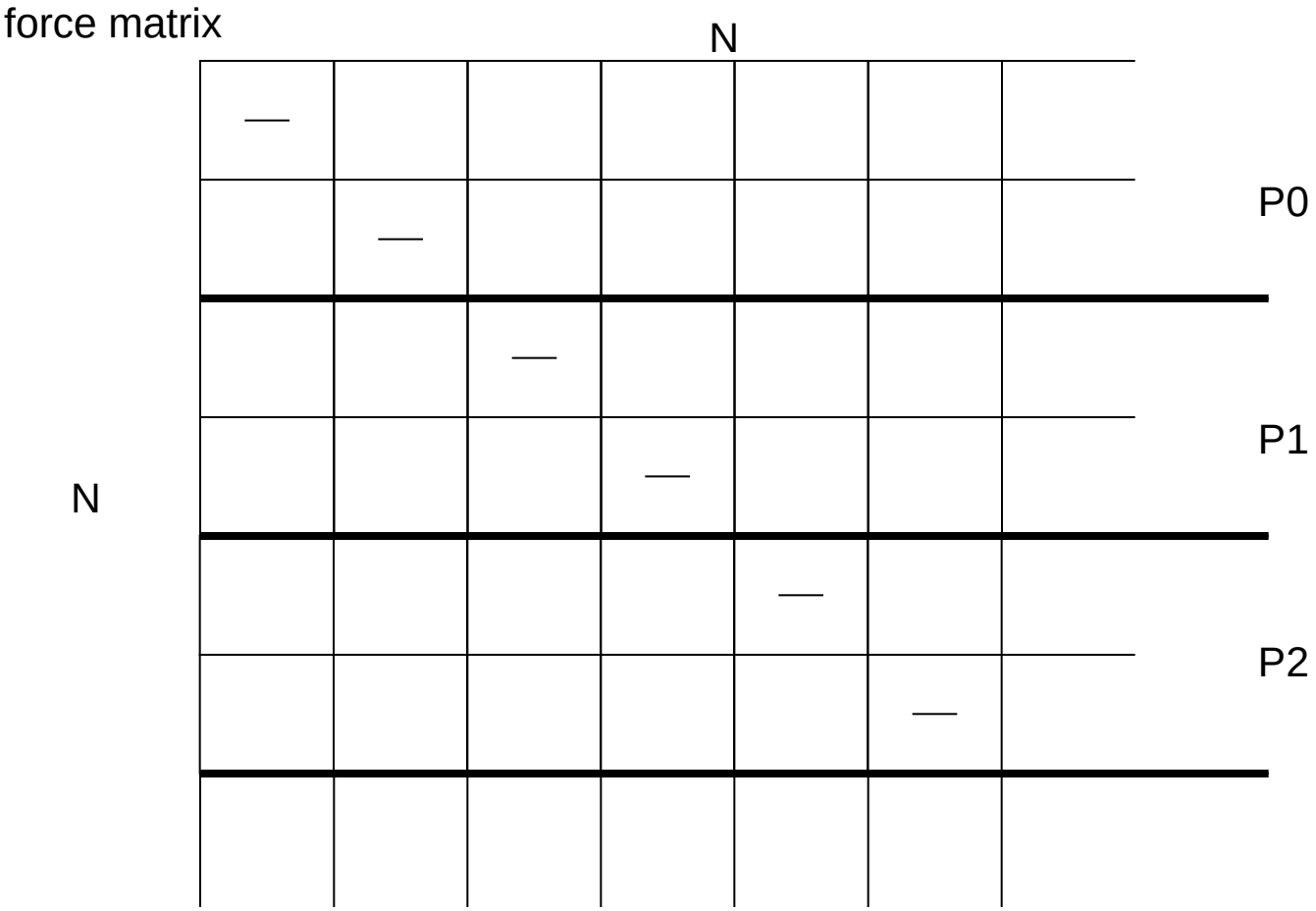
Assume pair-wise, short-range interactions between particles:

$$U = \sum_{i<j} U_{ij}$$

Serial code

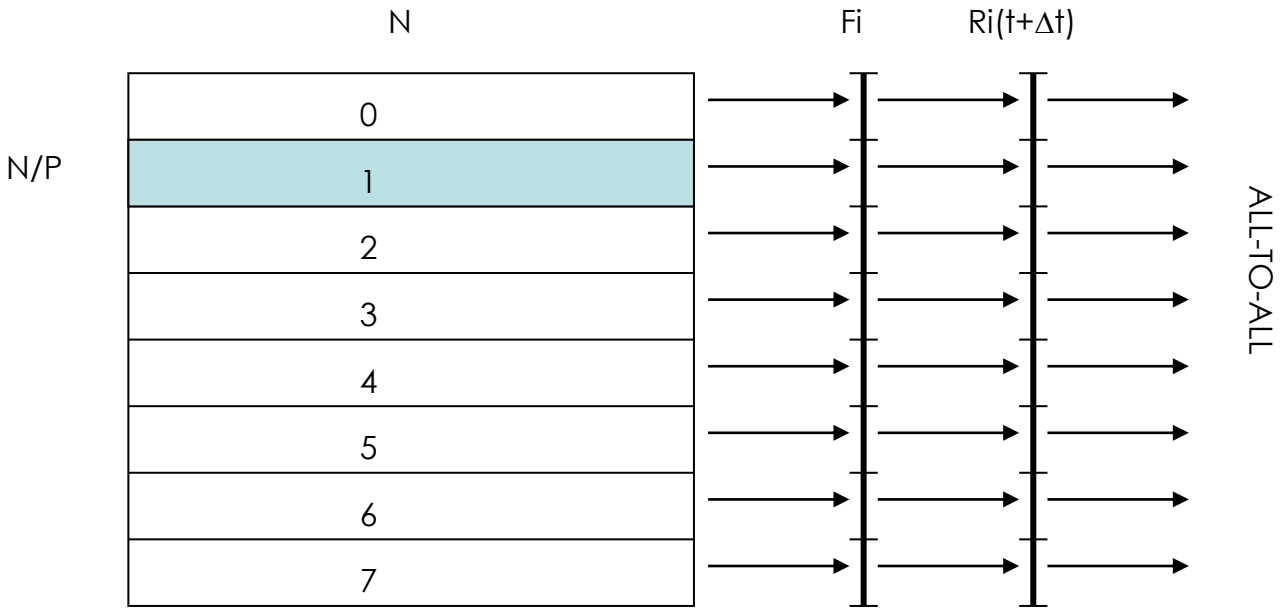
```
Utot=0.0
do i=1,N-1
   F(i) = 0.0
   do j=i+1,N
      rij=r(i)-r(j)
      Utot=Utot+Uij
      F(i)=F(i)+force(i,j)
   enddo
enddo
```

Interaction Matrix



- Double loop is equivalent to an interaction (e.g.force) matrix
- Can parallelise by replicating the data on each processor

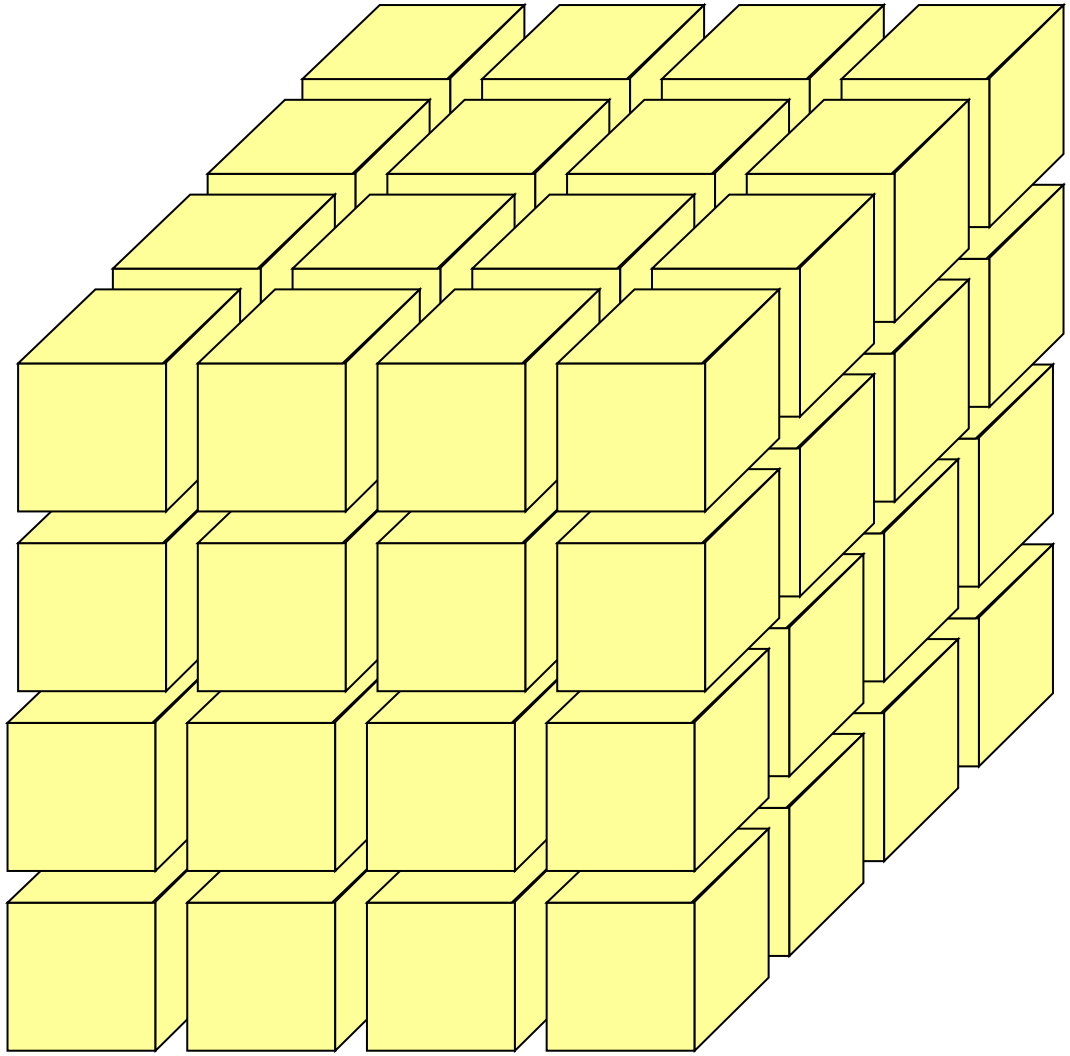
Particle decomposition



- In particle decomposition there is a normally at least one all-to-all communication step to gather the processor information
- In Molecular Dynamics codes (above) at least three global communications are required.

Particle decomposition (replicated data)

- Simple to code and scales reasonably well for small N.
- But communication scales as *O*(N) so at large N the all-to-all communication knocks performance – may not scale over 8-16 processors.
- Replicated data means memory requirements are also high.
- Possible to improve performance by using a block subdivision of the interaction matrix, but scaling is still limited.
- To do better we can exploit the *locality* of the interactions and use domain decomposition.



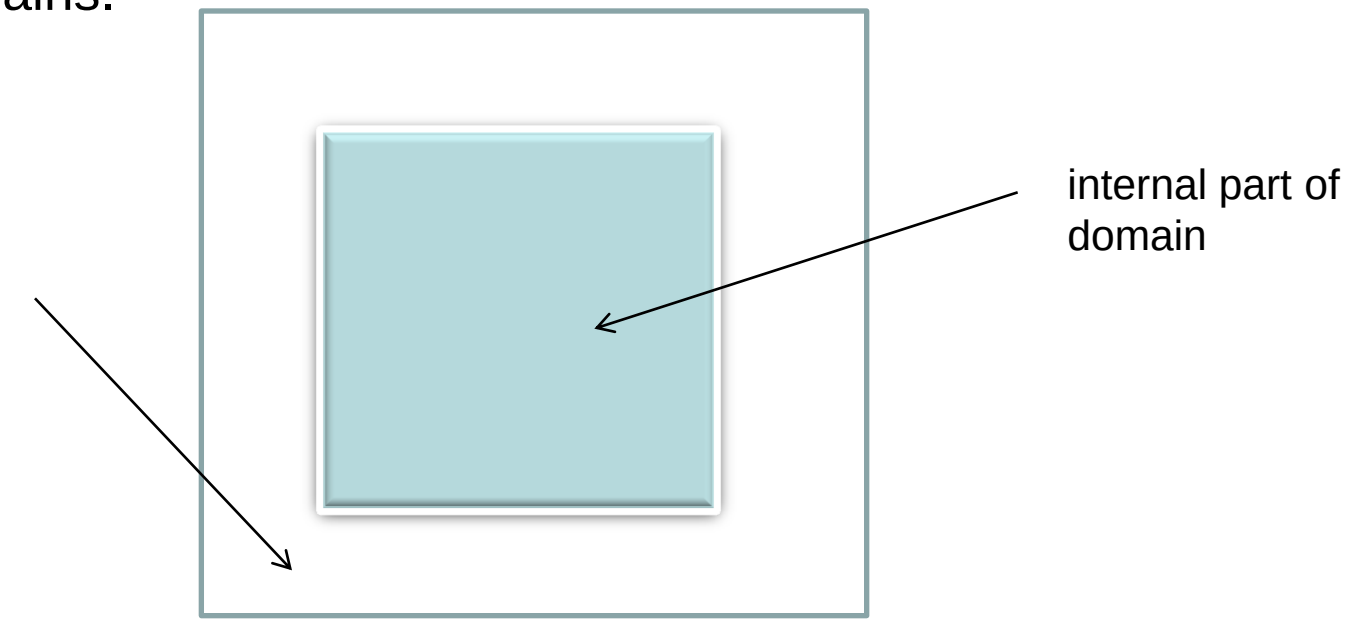
Divide 3d space up into domains and assign particles to domains. Also assign a processor to each domain.

Domain size usually chosen such that particles in one domain only interact with those in nearest-neighbour domains.

domain decomposition

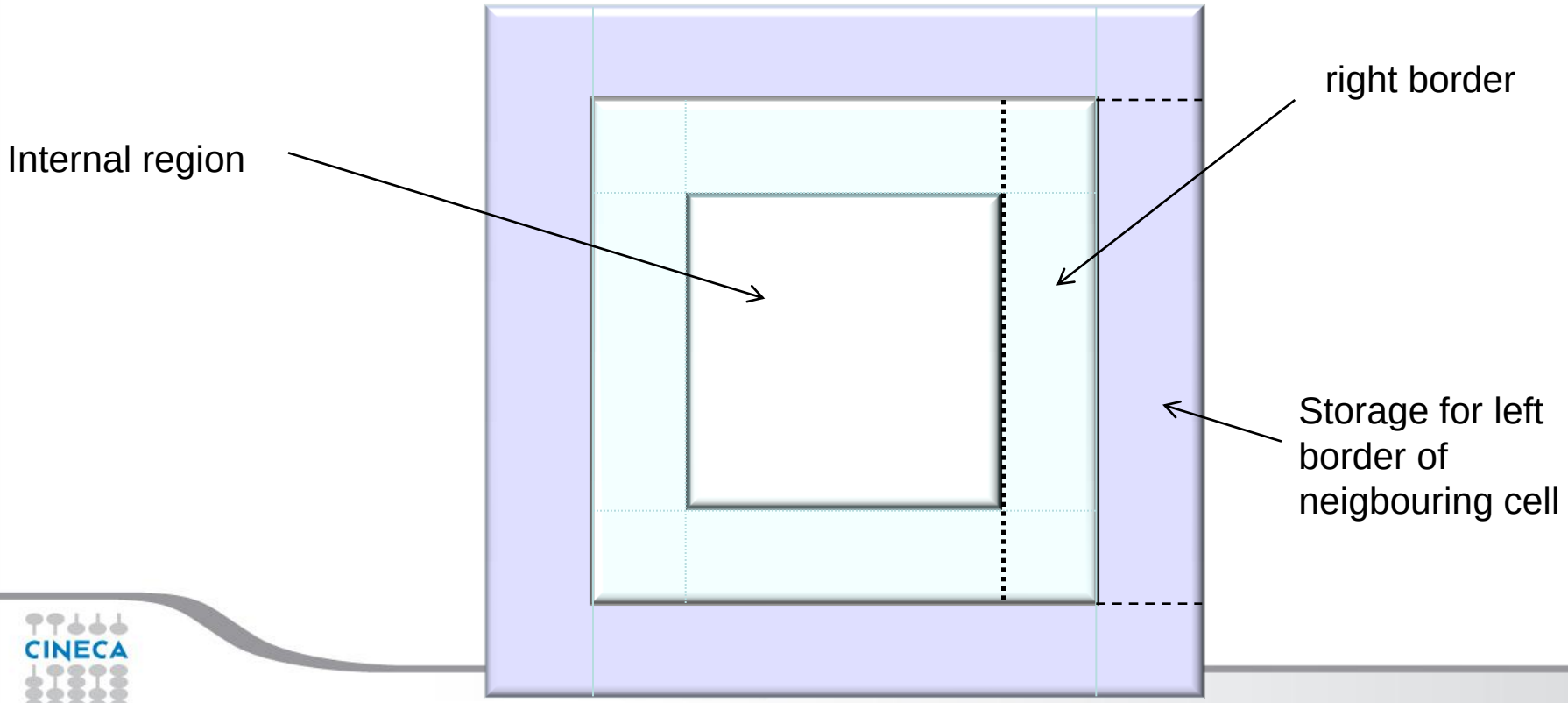
Each domain will have 2 types of particles:

1. Those which can be entirely managed by the processor, i.e. all the interactions are within the domain
2. Those for which the interactions *extend* outside the domain. Here data have to be sent/received to/from neighbouring domains.



domain decomposition

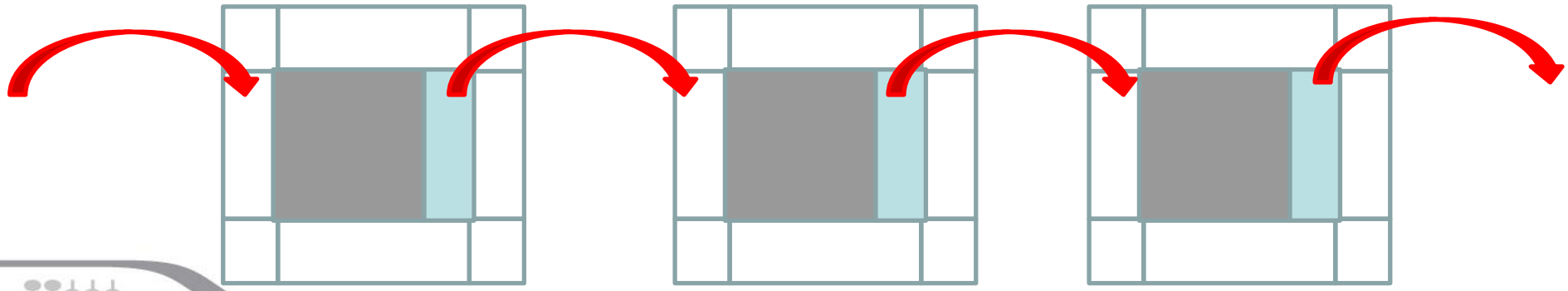
- The difficult part of DD is then to communicate data between a domain and its neighbours.
- Convenient for each processor to assign storage also for atoms in neighbouring regions within the cutoff *(some times call "ghost" or "halo"* regions).



domain decomposition - neighbour communication

- neighbour coords in each dimension conveniently exchanged via mpi_cart_shift and mpi_sendrecv calls
- First pass, x-direction, left to right,

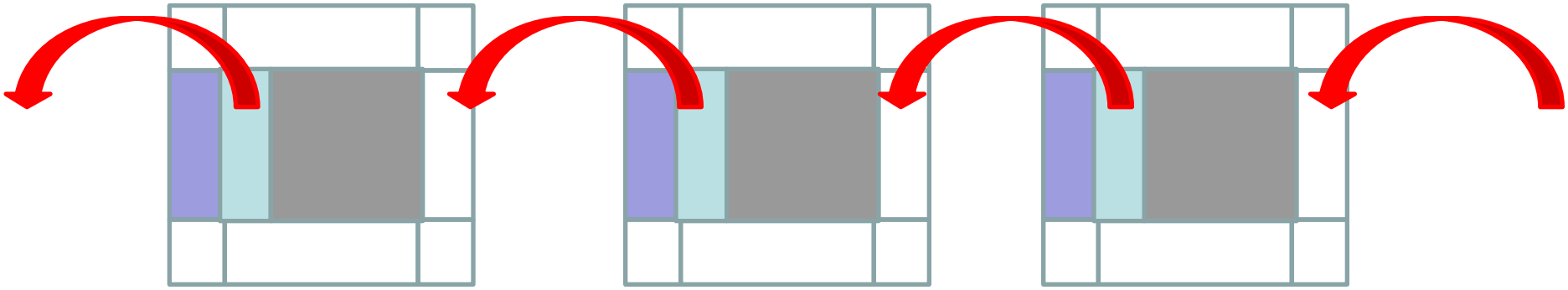
```
call mpi_cart_shift(mpi_box,1,1,proc_left,proc_right,ierror)

call mpi_sendrecv(right_side,nright,MPI_INTEGER,proc_right,0,
halo_left,nleft,MPI_INTEGER,proc_left,0,mpi_box,status,ierror)
```



domain decomposition

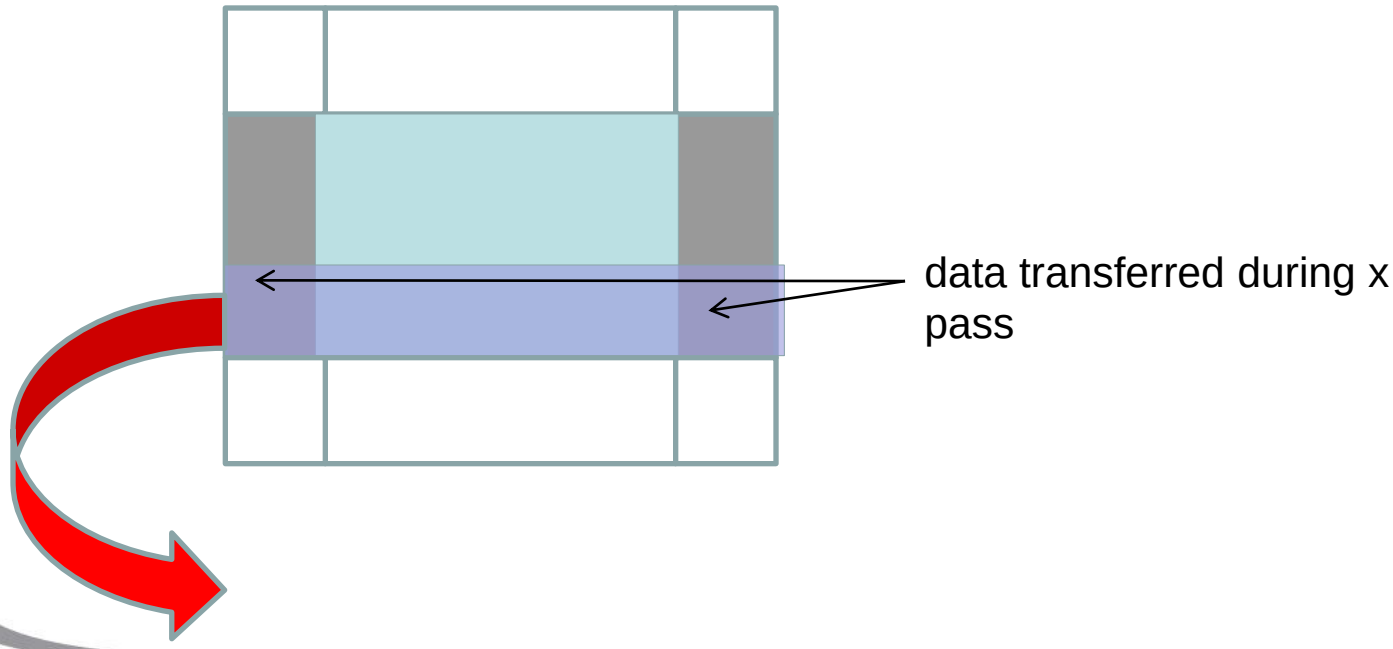
Then right to left

```
call mpi_sendrecv(left_side,nleft,MPI_REAL,proc_left,0,
halo_right,nright,MPI_REAL,proc_right,0,mpi_box,status,ier
ror)
```



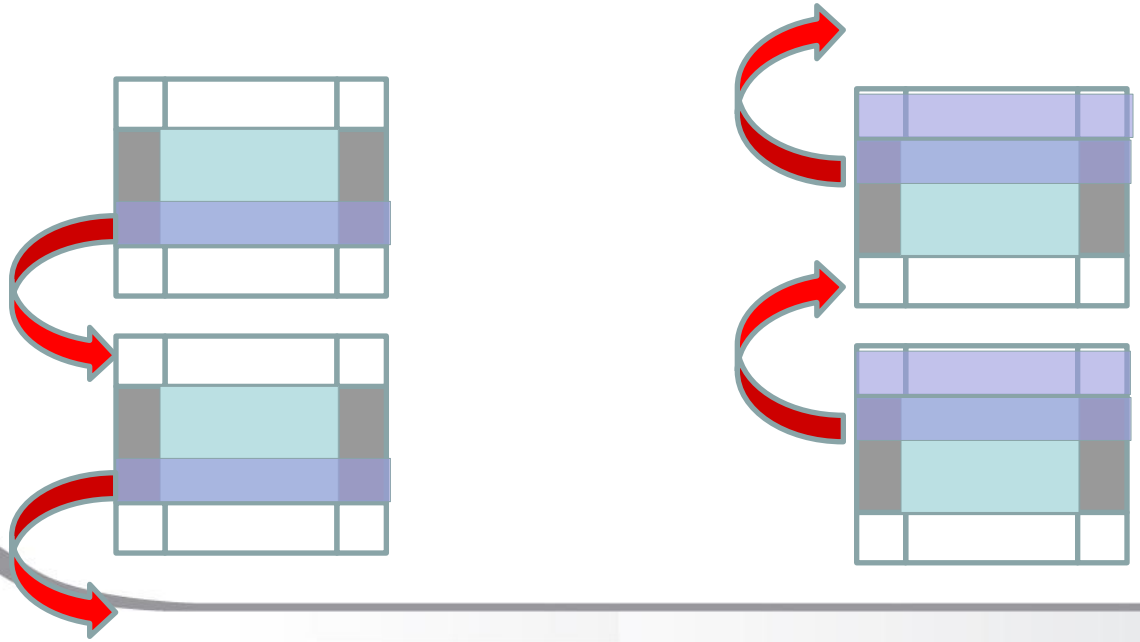
domain decomposition

We can repeat in the y direction but to ensure we transfer the corners we need to include data transferred in the x pass



domain decomposition

```
call mpi_cart_shift(mpi_box,0,1,proc_up,proc_down,ierror)

! top to bottom
call mpi_sendrecv(bottom_side,nlower,MPI_FLOAT,proc_down,0,
halo_top,ntop,MPI_REAL,proc_up,0,mpi_box,status,ierror)

! bottom to top
call mpi_sendrecv(top_side,ntop,MPI_REAL,proc_up,0,
halo_bottom,nbottom,MPI_REAL,proc_down,0,mpi_box,status,ier
ror
```



domain decomposition

- Similarly in the z direction, using data transferred in the previous y passes (which includes data transferred in x)

- Each processor now has enough information to calculate all the interactions in its domain.

particle and domain decomposition

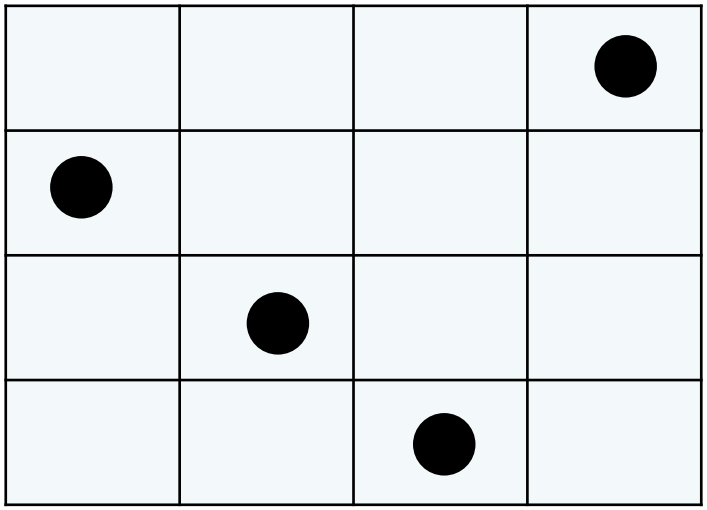
- Compared to PD (or Replicated Data), DD
  - Exploits the intrinsic **locality** , minimizing communications (no All-to-All) and memory required per processor
  - scalable, for large systems
  - can exploit MPI cartesian topology

Case study 1 – Game of Life

- A simple 2D cellular automata originally conceived by J. Conway in 1970.
- Based on a few simple rules, able to exhibit complex evolution depending on starting configuration and run time.
- Locality of interactions (i.e. state of neighbouring cells) implies good candidate for parallelization by domain decomposition

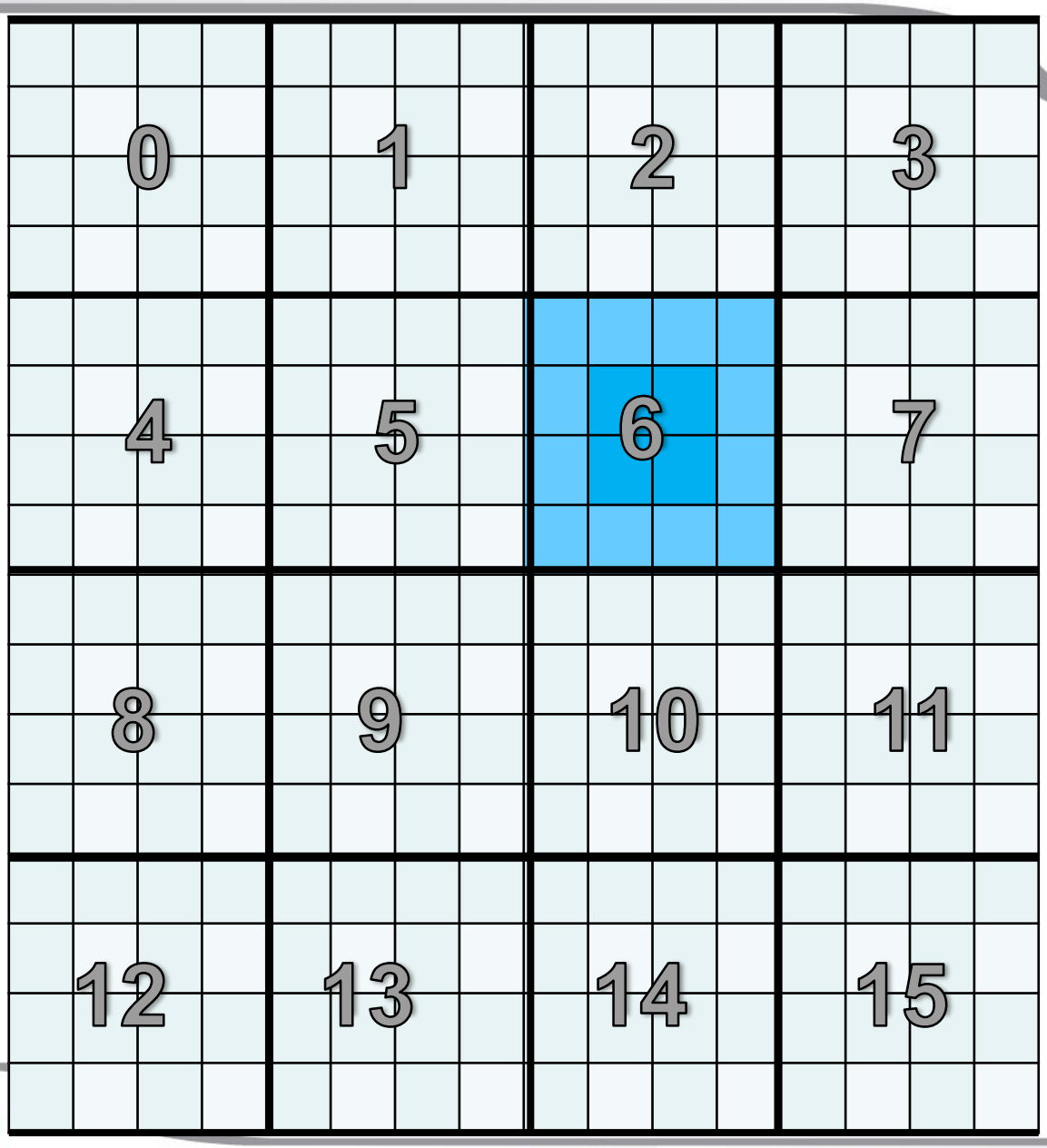
Case study 1 - Game of Life

- The system consists of a 2D grid of cells. Cells evolve as follows: in the next generation a cell will
  1. Be **dead** if the cell has < 2 live neighbours (lonely)
  2. **Stay the same** if has exactly 2 neighbours (content)
     1. **Be born** if the cell has exactly 3 live neighbours
  3. **Die** if > 3 live neighbours (overcrowding)



Game of Life - strategy

- Maintain two boards, one for the current generation and one for the next generation.
- Create a master-slave model: the master (e.g. rank 0) will generate the original configuration, collect results from other procs and write output to file.
- Partition the 2D array amongst the processors.
- Generate a cartesian topology.
- Each processor allocates storage for its own cells + halo regions (for neighbours).
- Procs update their own cells, then communicate boundaries to neighbouring cells. Calculate remaining cells.
- Master gathers data from procs, updates current board, writes to file → *next generation.*



Game of Life decomposition

Game of Life - implementation hints

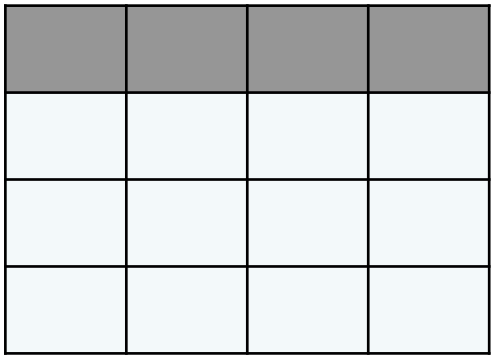
- cartesian topology

```
integer dlength(2),reorder
logical periods(0:1)
call mpi_init(ierror)
call mpi_comm_size(MPI_COMM_WORLD, size, ierror)

! Cartesian topology for grid

call mpi_dims_create(size,2,dlength,ierror)
periods(0)=.true.
periods(1)=.true.
reorder=1
call mpi_cart_create(MPI_COMM_WORLD,2,dlength,periods,
reorder,mpi_grid, ierror)
```

Game of Life - implementation hints

- MPI derived data types for transferring data to neighbours

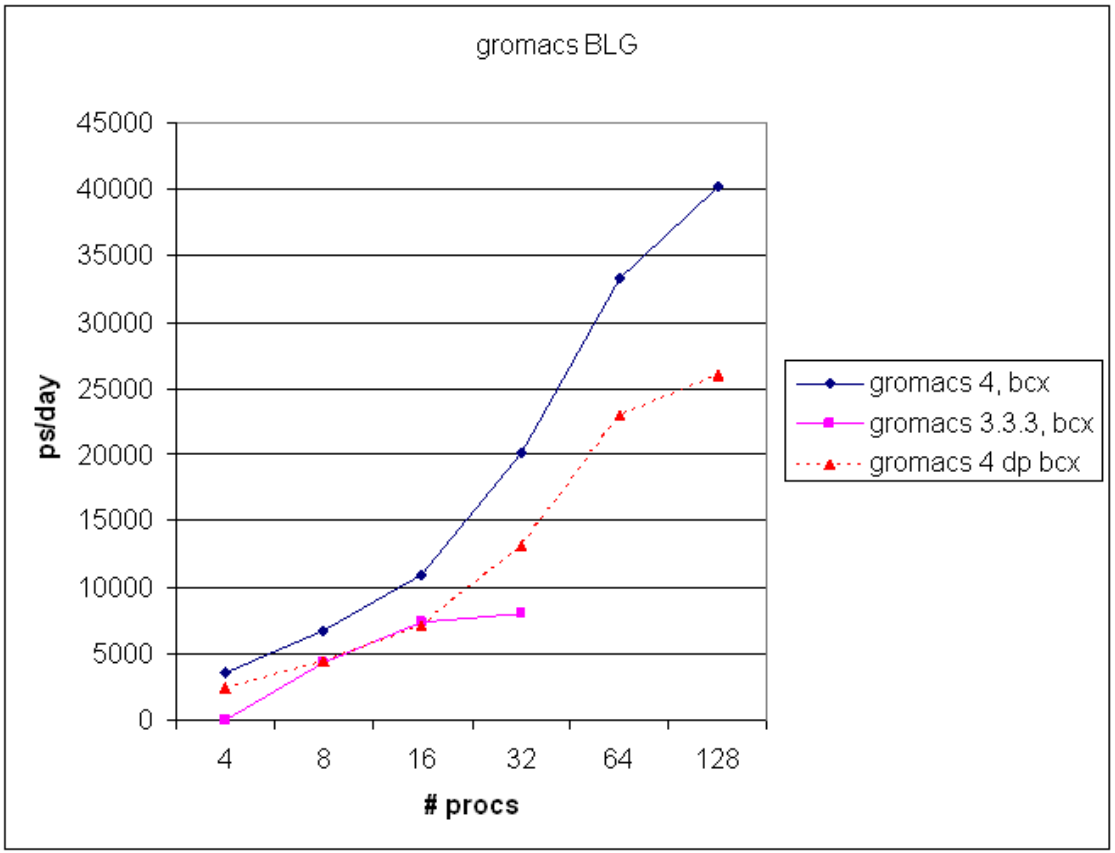
```
integer mpi_block, coltype, rowtype

! define a row type
call mpi_type_vector (local,1,nrow,MPI_INTEGER,rowtype,
ierror)
call mpi_type_commit(rowtype,ierror)

! find up and down neighbours
call
mpi_cart_shift(mpi_grid,0,1,proc_up,proc_down,ierror)

! send row to down proc, receive row data from up proc
call
mpi_sendrecv(locarray(nrow,1),1,rowtype,proc_down,0, &
edge_up,ncol,MPI_INTEGER,proc_up,0,mpi_grid,status,ierro
r)
```



Case study 2 – Classical molecular dynamics

- Molecular dynamics (MD) programs model physical or chemical systems by simulating the movements of interacting atoms or molecules.
- For realistic models, many tens of thousands or even millions of interacting atoms may need to be simulated.
- All common MD programs (e.g. GROMACS, NAMD, DL_POLY, etc) rely on DD for parallelisation.

Particle and domain decomposition comparison



- Gromacs v3.3 used particle/force decomposition as a parallel scheme.
- DD was introduced into Gromacs 4.x

final comments

- domain decomposition commonly used in computational chemistry, physics and astrophysics to distribute physical domain over processors by exploiting locality of interactions
- can be quite complex to program but MPI has many useful commands to simplify programming.