Programming with MPI
advanced point to point

Jan Thorbecke

Acknowledgments

- This course is partly based on the MPI courses developed by
  - Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
    http://www.hlrs.de/home/
    https://www.epcc.ed.ac.uk

- CSC – IT Center for Science Ltd.
  - https://www.csc.fi
  - https://research.csc.fi

- http://mpitutorial.com

Contents

- Communication groups
- Virtual topologies
- Cartesian Domain Decomposition
  - exercise: MPI_Cart
- MPI (derived) data types
  - exercise: Data_types

Communicators and Groups

- Create your own **communicator** for a subset of MPI-tasks
  - MPI_Comm_split()
- Create a MPI-**group** with specific MPI-tasks and create a communicator for this group.

Communicator Management

- Communicator Accessors
  - MPI_COMM_SIZE(…)
  - MPI_COMM_RANK(…)
- Communicator Constructors
  - MPI_COMM_CREATE(…)
  - MPI_COMM_SPLIT(…)
- Communicator Destructors
  - MPI_COMM_FREE(comm)

MPI_Comm_split

Create communicator for even task numbers:

color = myrank % 2

MPI_Comm_split(MPI_COMM_WORLD, color, key, &newcomm)

Tasks with the same color end-up in the same communication group (newcomm).

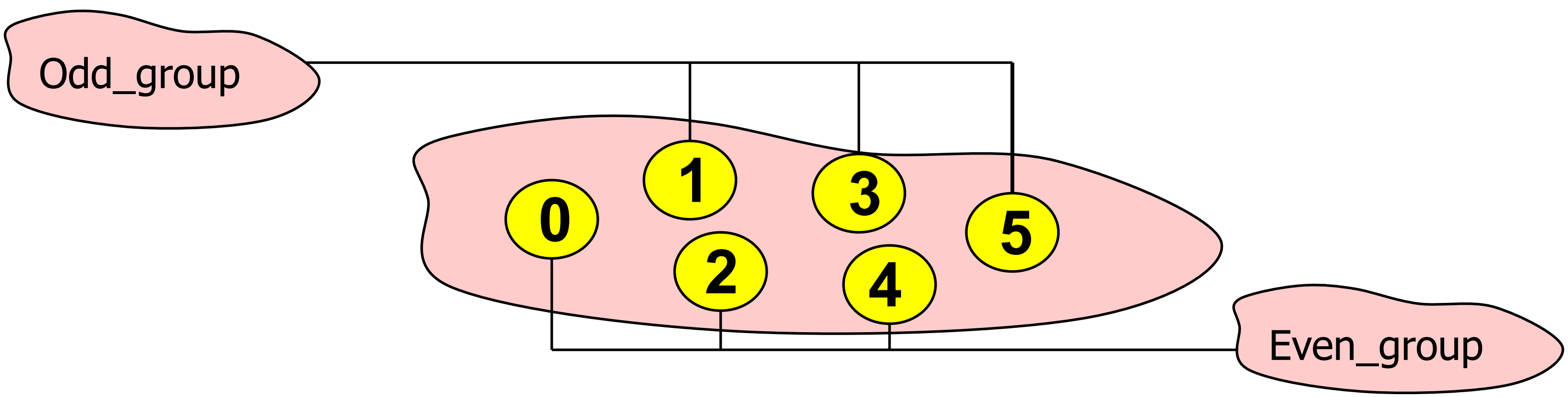
Key defines the ordering in the new communicator.

Group Management

- Group Accessors
  - MPI_Group_size(…)
  - MPI_Group_rank(…)
- Group Constructors
  - MPI_COMM_GROUP(…)
  - MPI_GROUP_INCL(…)
  - MPI_GROUP_EXCL(…)
- Group Destructors
  - MPI_GROUP_FREE(group)

Working with groups

- Select processes ranks to create groups
- Associate to these groups new communicators
- Use these new communicators as usual
- MPI_Comm_group(comm, group) returns in group the group associated to the communicator comm



For the previous example

Odd_ranks={1, 3, 5}, Even_ranks={0, 2, 4}

MPI_Comm_group(MPI_COMM_WORLD, Old_group)

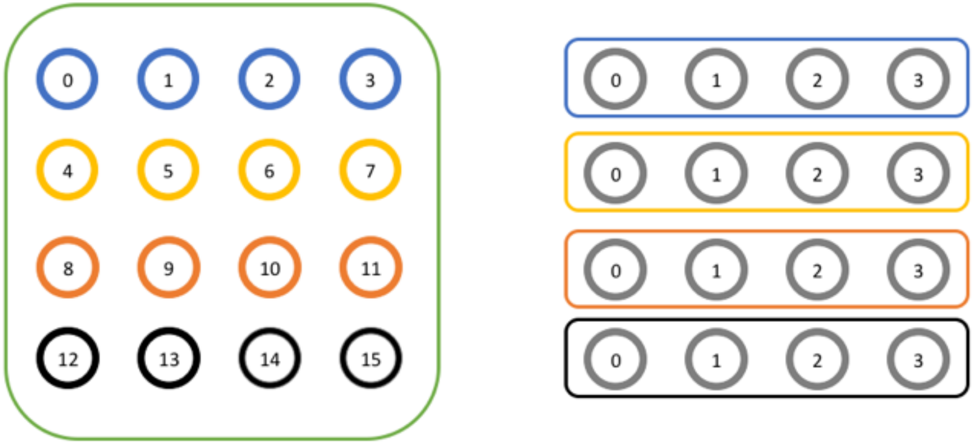
MPI_Group_incl(Old_group, 3, Odd_ranks,&Odd_group)
MPI_Group_incl(Old_group, 3, Even_ranks, &Even_group)

MPI_Comm_create(MPI_COMM_WORLD, Odd_group, Odd_Comm)
MPI_Comm_create(MPI_COMM_WORLD, Even_group,Even_Comm)

Exercise: GroupSplit

- Explains the use of new communicators and groups
- No programming needed follow the guidelines in the README.

Split a Large Communicator Into Smaller Communicators



Exercise: GroupSplit

- split.c
- split2.c
- groups.c

Virtual Topologies

Virtual topology

- For more complex mapping, mpi routines are available

  Global array A(1:3000, 1:4000, 1:500) = $6 \cdot 10^9$ words
  on 3 x 4 x 5 = 60 processors
  process coordinates 0..2, 0..3, 0..4

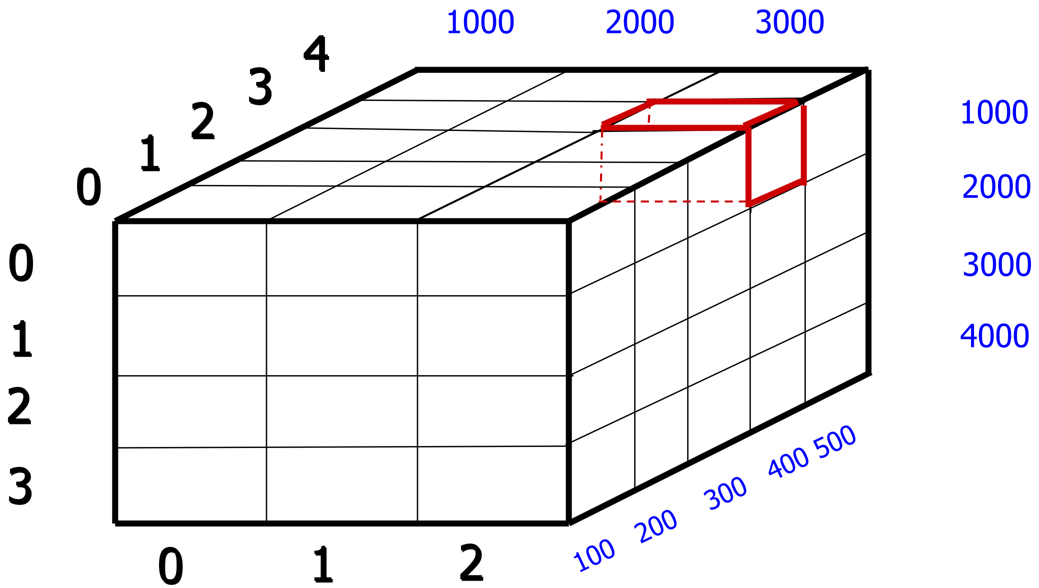
- example:
  on process $ic_0=2$, $ic_1=0$, $ic_2=3$ (rank=43)
  decomposition, A(2001:3000, 1:1000, 301:400)= $0.1 \cdot 10^9$ words

- process coordinates: handled with virtual Cartesian topologies

- Array decomposition: handled by the application program directly

Graphical representation

- Distribution of processes over the grid
- Distribution of the Global Array
- Coordinate (2, 0, 3) represents process number 43 (2*(4*5) + 0*5 + 3)
- It is being assigned the cube A(2001:3000, 1:1000, 301:400)



Communication Topologies

- Process topologies in MPI allow simple referencing of processes
- Cartesian and graph topologies are supported
- Process topology defines a new communicator
- MPI topologies are virtual
  - no relation to the physical structure of the computer
  - data mapping "more natural" only to the programmer
- Usually no performance benefits
  - But code becomes more readable

How to use a Virtual Topology

- Creating a topology produces a new communicator.

- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.

Topology Types

- Cartesian Topologies
  - each process is connected to its neighbour in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course,
    communication between any two processes is still allowed.
- Graph Topologies
  - general graphs,
  - not covered here.

Creating a Cartesian Virtual Topology

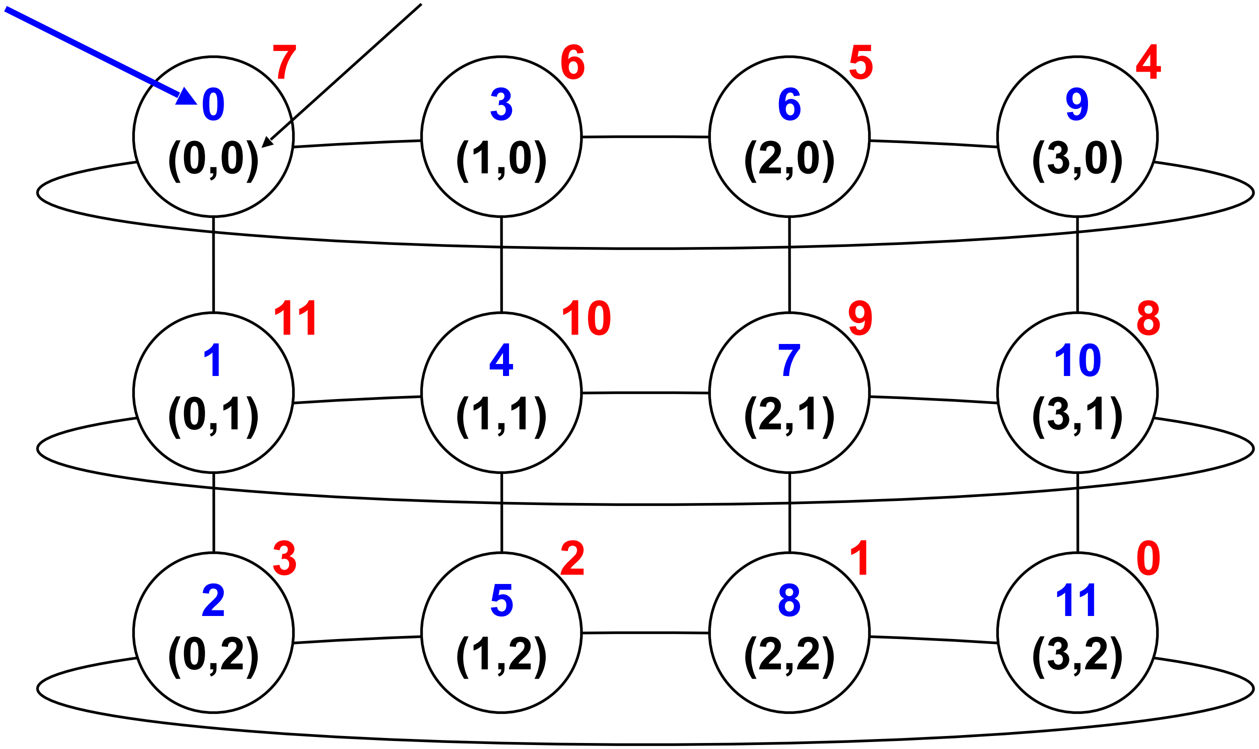
- New communicator with processes ordered in a Cartesian grid

```
MPI_Cart_create(oldcomm, ndims, dims,
  periods, reorder, newcomm)
```

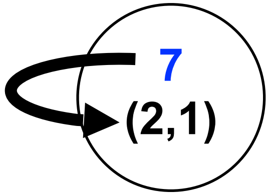
– **oldcomm**	communicator
– **ndims**	dimension of the Cartesian topology
– **dims**	integer array (size ndims) that defines the number of processes in each dimension
– **periods**	array that defines the periodicity of each dimension
– **reorder**	is MPI allowed to renumber the ranks
– **newcomm**	new Cartesian communicator

Example – dimensional Cylinder

- Ranks and Cartesian process coordinates in comm_cart
- Ranks in comm and comm_cart may differ, if <u>reorder = 1 or .TRUE.</u>
- This reordering can allow MPI to optimise communications



Cartesian Mapping Functions



Mapping
ranks to
process grid coordinates

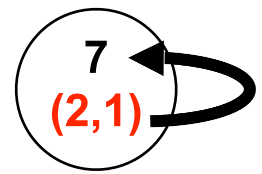
- Translate a rank to coordinates

`MPI_Cart_coords(comm, rank, maxdim, coords)`

  - **comm** Cartesian communicator
  - **rank** rank to convert
  - **maxdim** dimension of *coords*
  - **coords** coordinates in Cartesian topology that corresponds to *rank*

Cartesian Mapping Functions

Mapping
process grid coordinates
to ranks



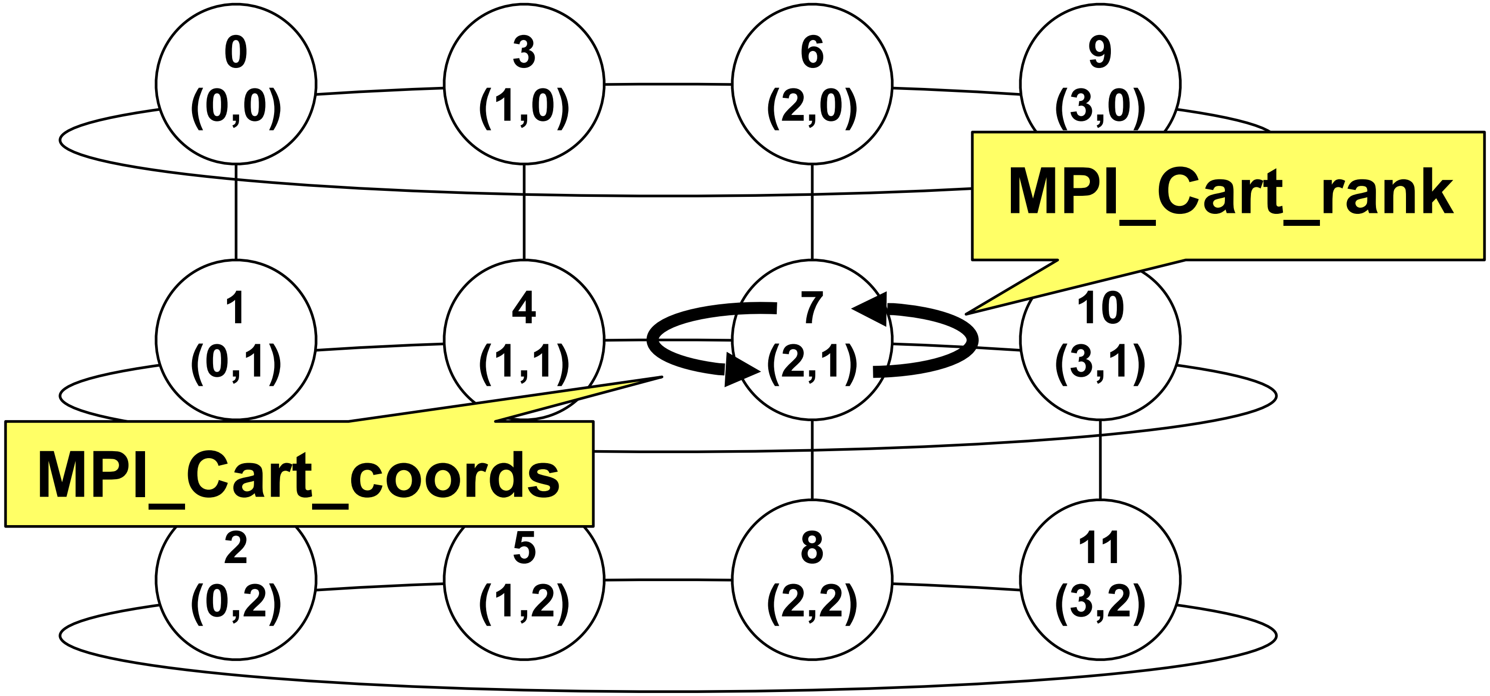
- Translate a set of coordinates to a rank

`MPI_Cart_rank(comm, coords, rank)`

- – **comm** Cartesian communicator
- – **coords** array of coordinates
- – **rank** a rank corresponding to *coords*

Own coordinates

- Each process gets its own coordinates with
  MPI_Comm_rank(comm_cart, my_rank, ierror)
  MPI_Cart_coords(comm_cart, my_rank, maxdims, my_coords, ierror)



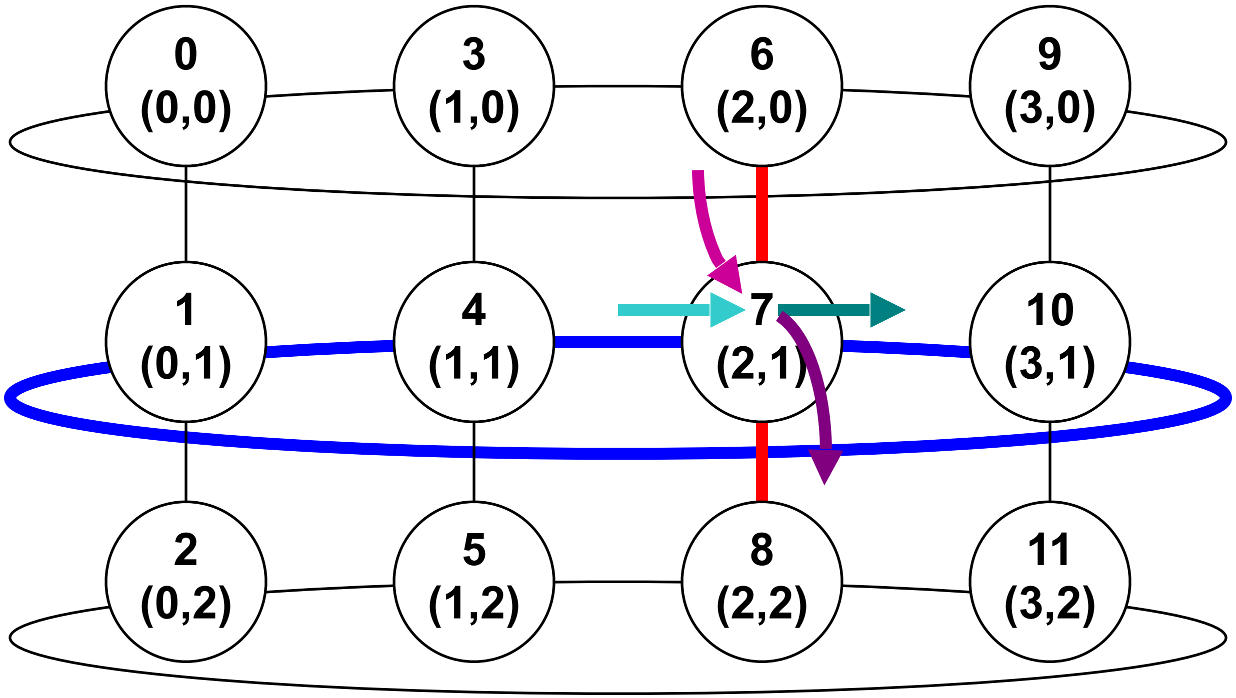
Cartesian neighbours

- Counting ”hops” in the Cartesian grid to allow for e.g. elegant nearest-neighbor communication

```
MPI_Cart_shift(comm, direction, displ,
  source, dest)
```

- **comm** Cartesian communicator
- **direction** shift direction (e.g. 0 or 1 in 2D)
- **displ** shift displacement (1 for next cell etc, <0 for “down”/”left” directions)
- **source** rank of source process
- **dest** rank of destination process
- With non-periodic grid, *source* or *dest* can land outside of the grid; then `MPI_PROC_NULL` is returned

MPI_Cart_shift – Example



invisible input argument: my_rank of the running process executing:

```
  MPI_Cart_shift( cart, direction, displace, prev, next)
     example on                0             +1       4     10
     process rank=7            1             +1       6      8
                               0             -1      10      4
```

Exercise: MPI_Cart

- Do I have to make a map of MPI processes myself?
- You can use MPI_Cart to create a domain decomposition for you.
- Exercise sets up 2D domain with one
  - periodic and
  - non-periodic boundary condition
- README for instructions

answer 16 MPI tasks

```
PE-  6: x=1, y=2: Sum = 32 neighbors: left=2 right=10 top=5 bottom=7
PE-  4: x=1, y=0: Sum = 24 neighbors: left=0 right=8 top=-1 bottom=5
PE-  2: x=0, y=2: Sum = 32 neighbors: left=14 right=6 top=1 bottom=3
PE-  7: x=1, y=3: Sum = 36 neighbors: left=3 right=11 top=6 bottom=-1
PE-  0: x=0, y=0: Sum = 24 neighbors: left=12 right=4 top=-1 bottom=1
PE-  5: x=1, y=1: Sum = 28 neighbors: left=1 right=9 top=4 bottom=6
PE-  1: x=0, y=1: Sum = 28 neighbors: left=13 right=5 top=0 bottom=2
PE-  3: x=0, y=3: Sum = 36 neighbors: left=15 right=7 top=2 bottom=-1
PE- 14: x=3, y=2: Sum = 32 neighbors: left=10 right=2 top=13 bottom=15
PE-  9: x=2, y=1: Sum = 28 neighbors: left=5 right=13 top=8 bottom=10
PE-  8: x=2, y=0: Sum = 24 neighbors: left=4 right=12 top=-1 bottom=9
PE- 11: x=2, y=3: Sum = 36 neighbors: left=7 right=15 top=10 bottom=-1
PE- 15: x=3, y=3: Sum = 36 neighbors: left=11 right=3 top=14 bottom=-1
PE- 10: x=2, y=2: Sum = 32 neighbors: left=6 right=14 top=9 bottom=11
PE- 12: x=3, y=0: Sum = 24 neighbors: left=8 right=0 top=-1 bottom=13
PE- 13: x=3, y=1: Sum = 28 neighbors: left=9 right=1 top=12 bottom=14
```

exercise: MPI_Cart

periodic boundaries

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Domain Decomposition

for example: wave equation in a 2D domain.

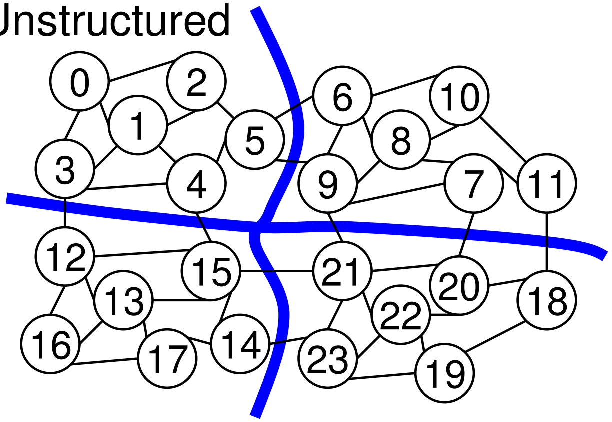
1. Break up the domain into blocks.
2. Assign blocks to MPI-processes one-to-one.
3. Provide a "map" of neighbours to each process.
4. Write or modify your code so it only updates a single block.
5. Insert communication subroutine calls where needed.
6. Adjust the boundary conditions code.
7. Use "guard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

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

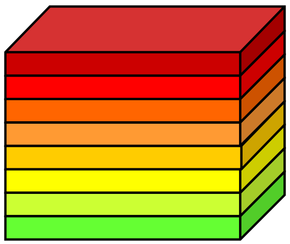
Unstructured



Examples with 4 sub-domains

-1- Break up domain in blocks

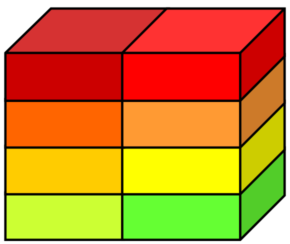
Splitting in



- **one** dimension:

  communication

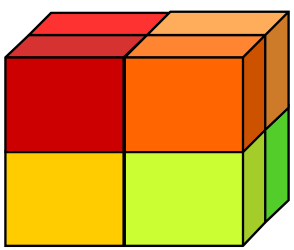
  $= n^2 * 2 * w * 1$



- **two** dimensions:

  communication

  $= n^2 * 2 * w * 2 / p^{1/2}$



- **three** dimensions:

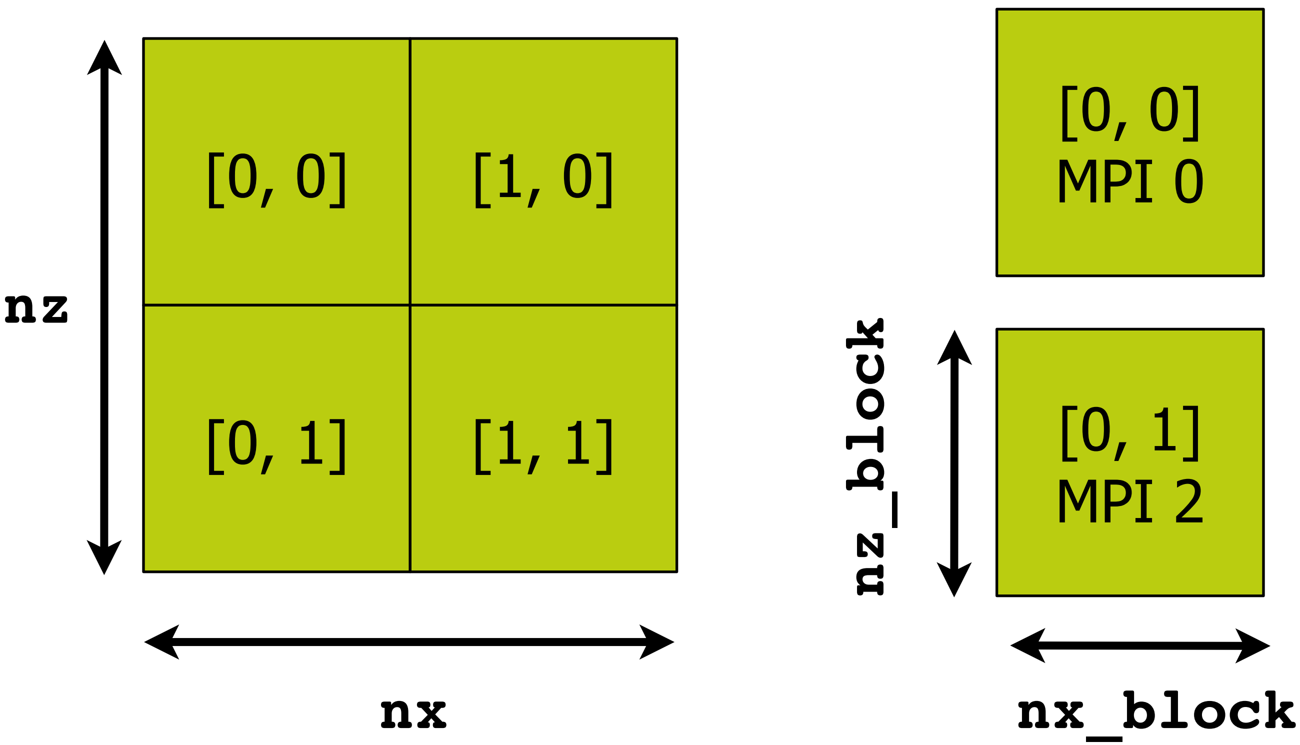
  communication

  $= n^2 * 2 * w * 3 / p^{2/3}$

optimal for p>11

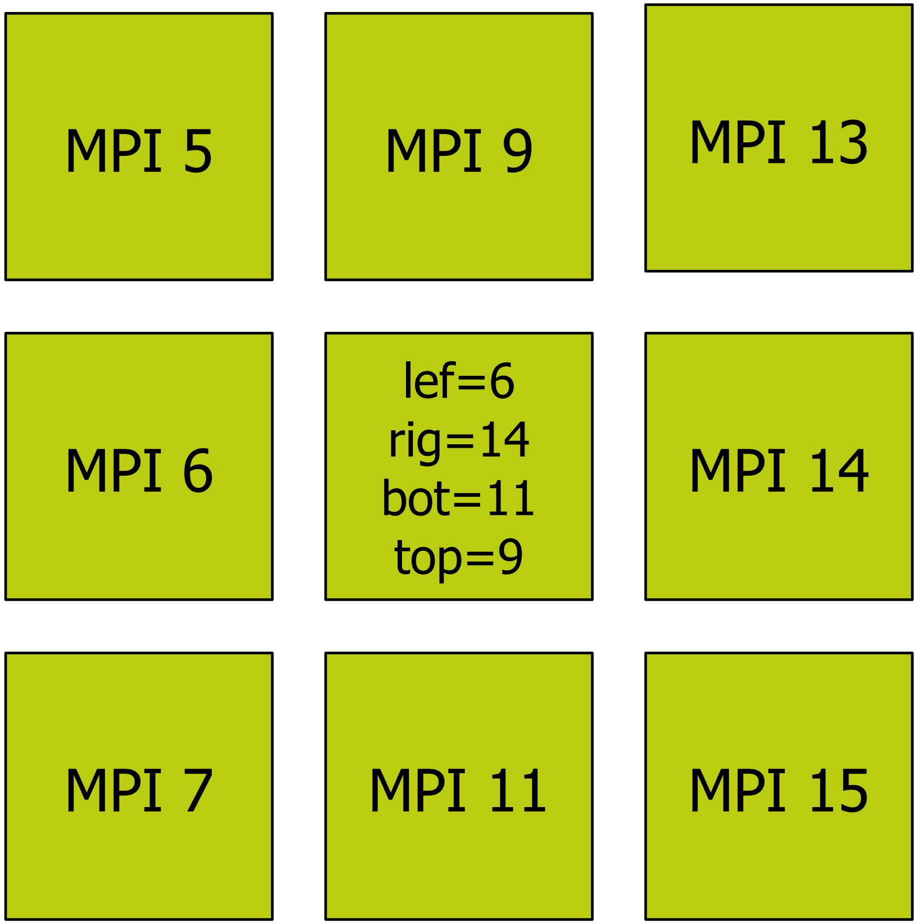
w=width of halo
n^3 =size of matrix
p = number of processors
cyclic boundary
—> two neighbors in each direction

-2- Assign blocks to MPI processes



nx_block=nx/2
nz_block=nz/2

-3- Make a map of neighbours



-4- Write code for single MPI block

```
for (ix=0; ix<nx_block; ix++) {
    for (iz=0; iz<nz_block; iz++) {
        vx[ix][iz] -= rox[ix][iz]*(
         c1*(p[ix][iz]   - p[ix-1][iz]) +
         c2*(p[ix+1][iz] - p[ix-2][iz]));
        }
    }
```

-5- Communication calls

- Examine the algorithm
  - Does it need data from its neighbours?
  - Yes? => Insert communication calls to get this data

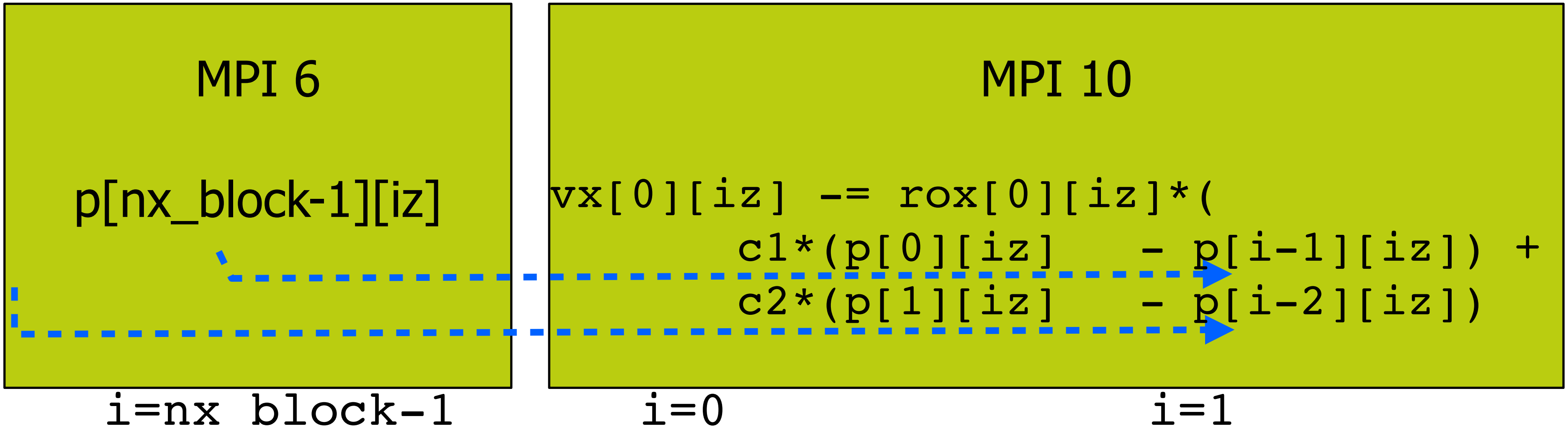
- Need p[-1][iz], p[-2][iz], p[nx_block+1][iz] for vx[0][iz]

- p[-1][iz] corresponds to p[nx_block-1][iz] from left neighbour
- p[-2][iz] corresponds to p[nx_block-2][iz] from left neighbour
- p[nx_block+1][iz] corresponds to p[0][iz] from right neighbour

- Get this data from the neighbours with MPI.

communication

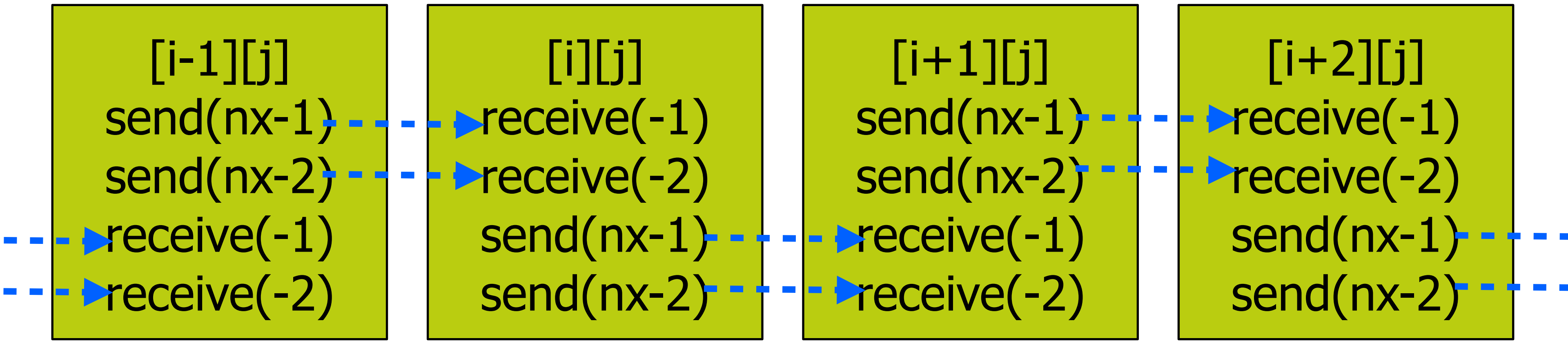
```
for (ix=0; ix<nx_block; ix++) {
    for (iz=0; iz<nz_block; iz++) {
        vx[ix][iz] -= rox[ix][iz]*(
         c1*(p[ix][iz]   - p[ix-1][iz]) +
         c2*(p[ix+1][iz] - p[ix-2][iz]));
        }
    }
```



pseudocode

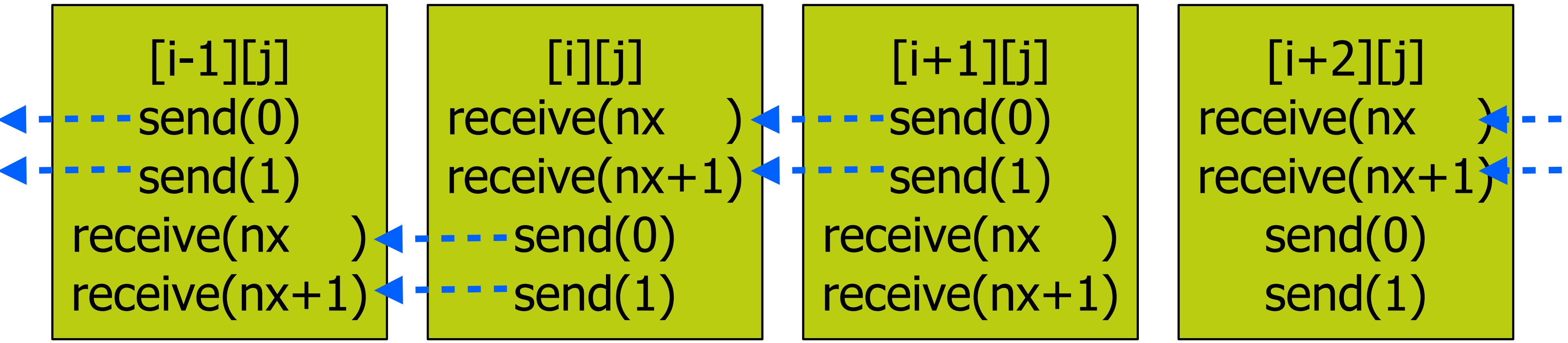
```
MPI_Send(&p[nx_block-2][iz], .., right, ..);
MPI_Receive(&p[-2][iz], .., left, ..);

MPI_Send(&p[0][iz], .., left, ..);
MPI_Receive(&p[nx_block][iz], .., right, ..);

        vx[ix][iz] -= rox[ix][iz]*(
         c1*(p[ix][iz]   - p[ix-1][iz]) +
         c2*(p[ix+1][iz] - p[ix-2][iz]));
```

Update of all blocks [i][j]




```
 mpi_sendrecv(
p(nx-1:nx-2), 2 ,..., my_neighbor_right,...
p(-2:-1),     2 ,..., my_neighbor_left,...)
```

Update of all blocks [i][j]

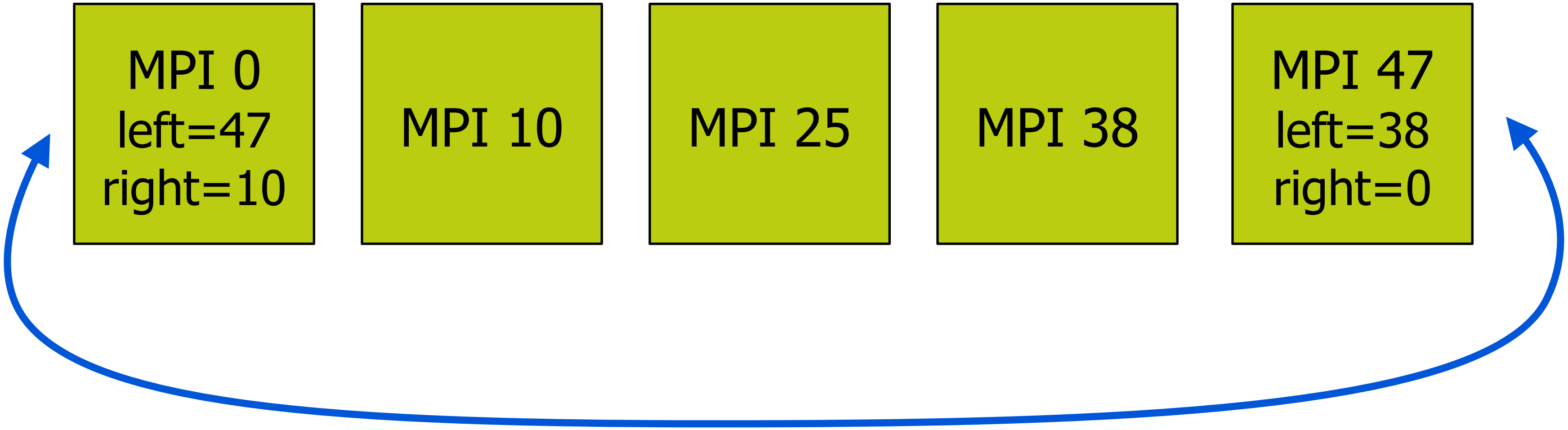



```
 mpi_sendrecv(
p(0:1),      2 ,..., my_neighbor_left,...
p(nx:nx+1), 2 ,..., my_neighbor_right,…)
```

-6- Domain boundaries

- What if my block is at a physical boundary?
- Periodic boundary conditions
- Non-periodic boundary conditions

Periodic Boundary Conditions



Non-Periodic Boundary Conditions

1. Avoid sending or receiving data from a non-existent block.
   MPI_PROC_NULL

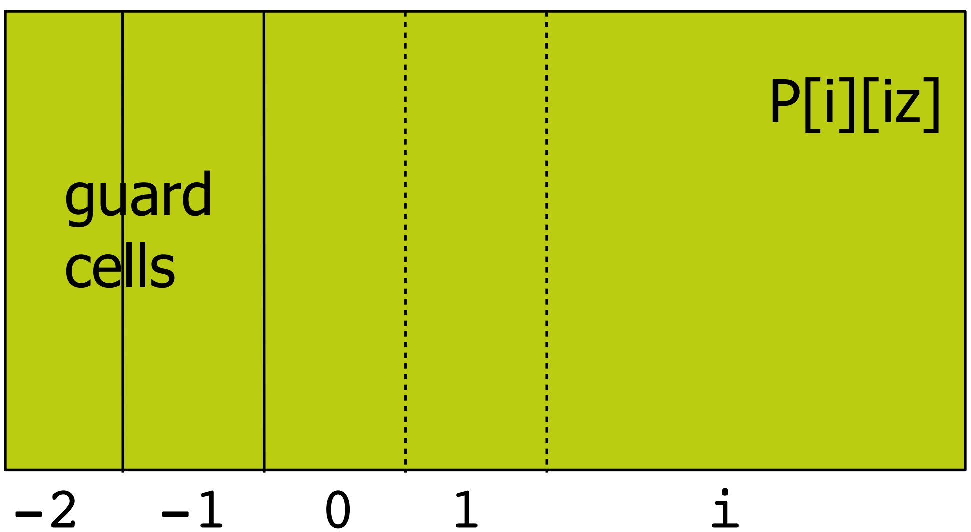
```
if (my_neighbor_right != MPI_PROC_NULL) {
    MPI_Send(…my_neighbor_right…)
```

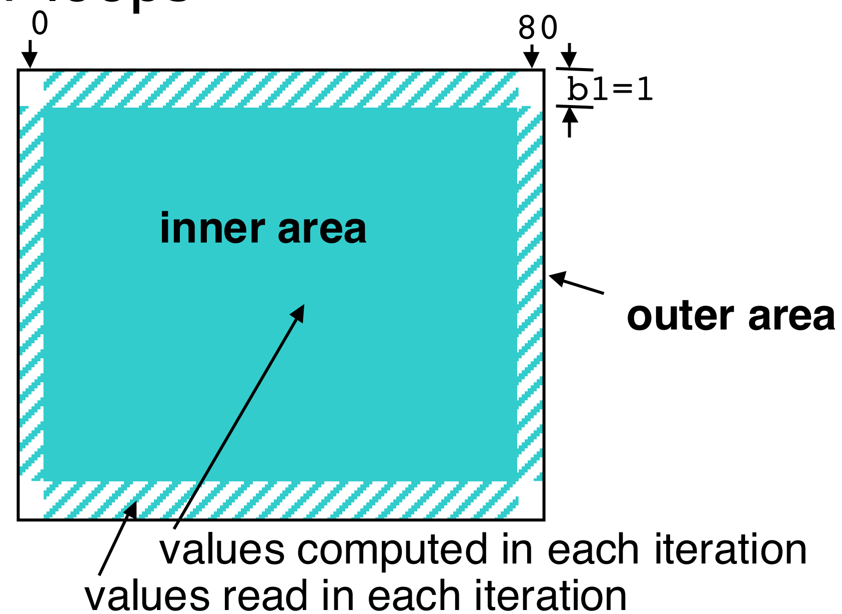
2. Apply boundary conditions.

```
if (my_neighbor_right == MPI_PROC_NULL) {
      [ APPLY APPROPRIATE BOUNDARY CONDITIONS ]
```

-7- guard cells or halo

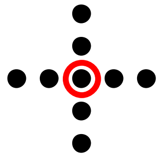
- Allocate local block including overlapping areas to store values received from neighbor communication.
- This allows tidy coding of your loops



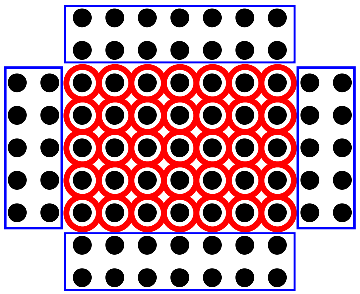


Halo areas

- Stencil:
  - To calculate a new grid point (○),
    old data from the stencil grid points (•) are needed
    - **E.g., 9 point stencil**



- Halo
  - 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

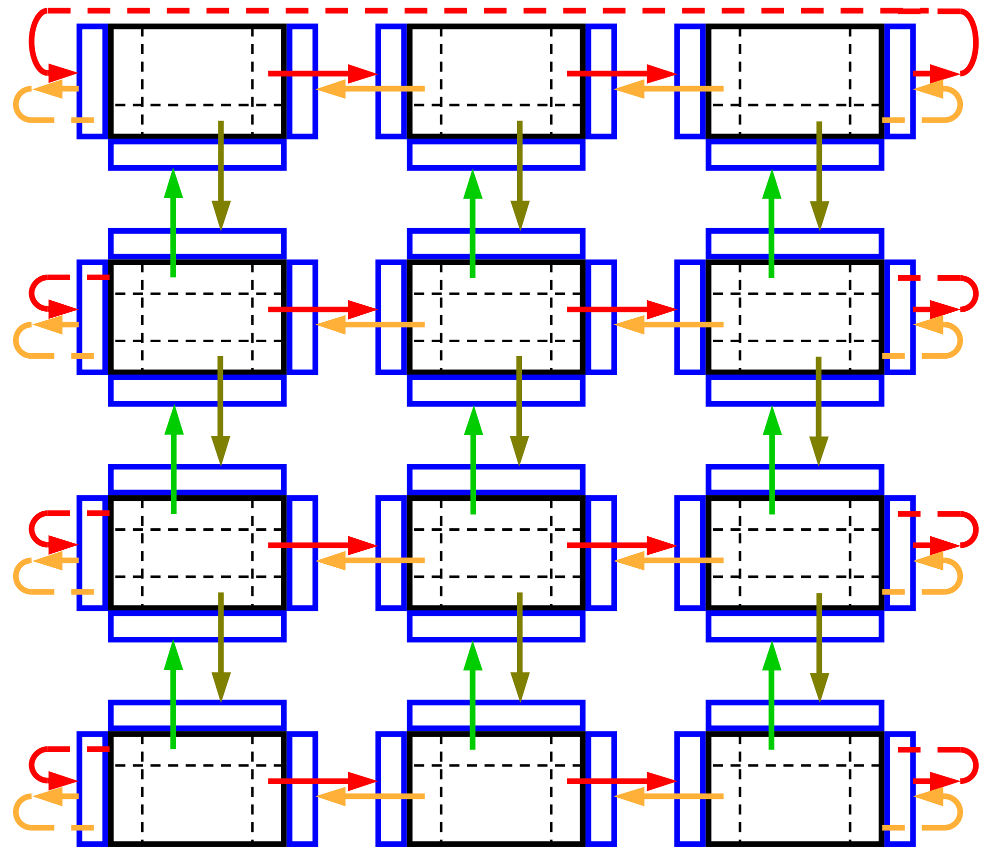


Communication to Halo-area

One iteration in the

- **serial** code:
  - $X_{\mathbf{new}} = \text{function}(x_{\mathbf{old}})$
  - $X_{\mathbf{old}} = x_{\mathbf{new}}$

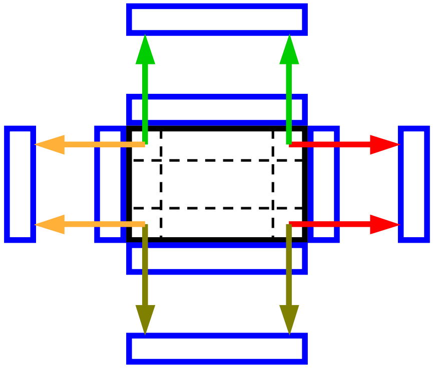
- **parallel** code:
  - Update halo
    [**=Communication, e.g., with**
    4 x MPI_Sendrecv]
  - $X_{\mathbf{new}} = \text{function}(x_{\mathbf{old}})$
  - $X_{\mathbf{old}} = x_{\mathbf{new}}$

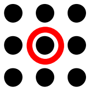


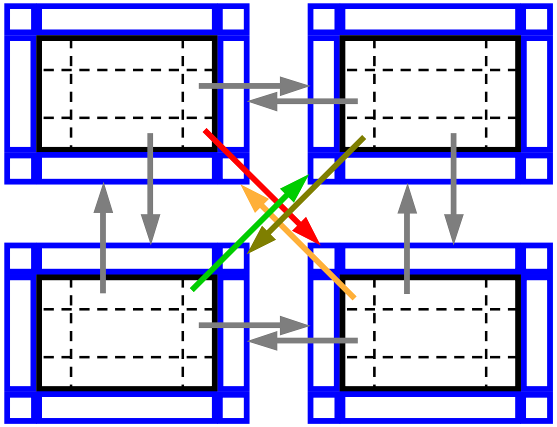
Examples with 12 sub-domains and horizontally cyclic boundary conditions
→ communication around rings

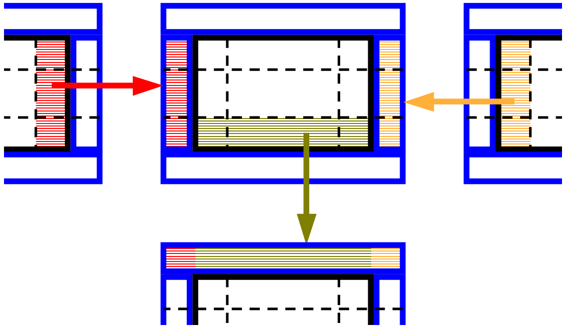
Corners

- MPI non-blocking send must not send inner corner data into more than one direction
  - Use MPI_Sendrecv
  - Or non-blocking MPI_Irecv



- Stencil with diagonal point, e.g., 
  - i.e., halos include corners →→→ substitute small corner messages:
    - **one may use 2-phase-protocol:**
    - **normal horizontal halo communication**
    - **include corner into vertical exchange**





Memory use increases with halo

- Example 1000x1000, N tasks, H square width of the task, h halo-points. The guard size per domain is
  - halosize = 4*h*H+4*corners(h*h)
- Halosize limits minimum task size: H >= 2h, hence limits maximum number of tasks N <= (Nx/2h)**2
- N=1089=33*33 then H=1000/33 ~ 30, h=8
  - halosize = 4*8*30+4*8*8 = 1216
  - actual work is 30*30

Demo code 2.5D finite-difference

- using 3DFD/3dfd.c
  - MPI_Dims_create() can help find a suitable decomposition.
  - MPI_Cart
  - MPI_IO
  - Hybrid MPI + OpenMP
- Code not yet fully functional for general use:
  - 2.5D; read in 2D model and invariant in y-dimension
  - PML not yet added (lots of book-keeping)
  - add input parameters for source and receiver positions
  - includes MPI-IO: will be covered soon
  - Good learning project to learn and use MPI ;-)

Derived datatypes

- To sent non-contiguous elements in memory in one MPI call.
- Useful to communicate halo-areas in domain decomposition

Sending a row of a Matrix

```
for (int row=0; i<N; i++) {
   MPI_Send(&m[row*N+col], 1, MPI_DOUBLE, peer, tag, comm);
}
```

```
double* buf = malloc(N*sizeof(double));
for (int row=0; i<N; i++) {
   buf[row] = m[row*N+col];
}
MPI_Send(buf, N, MPI_DOUBLE, peer, tag, comm);
```

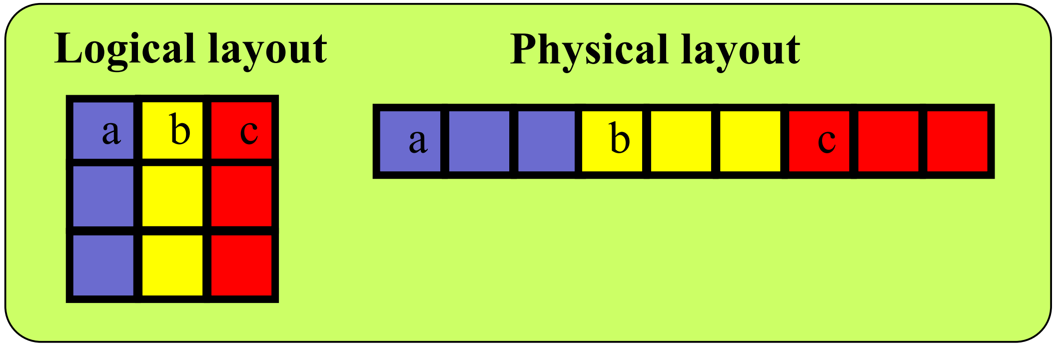
```
MPI_Datatype newtype;
MPI_Type_vector(N,blocklen,N,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Send(m, 1, newtype, peer, tag, comm);
```

MPI Datatypes

- Description of the memory layout of the buffer
  - for sending
  - for receiving
- Basic types
- Derived types
  - Vectors, structs, others
  - Built from existing datatypes

Example: sending row in Fortran

- Fortran uses column-major ordering, so row of a matrix is not contiguous in memory



In N x M matrix,
each element of a row is
separated by N elements

- Several options for sending a row:
  - Create datatype and send all data with one send command
  - Use several send commands for each element of a row
  - Copy data to temporary buffer and send that with one send command

Creating derived data-type

- A new datatype is created from existing ones with a datatype constructor
  - Several different commands for different special cases
  - MPI_Type_Vector, ...
- A new datatype must be committed before using it.

```
MPI_Type_commit(newtype)
newtype   the new type to commit
```

- A type should be freed after it is no longer needed.

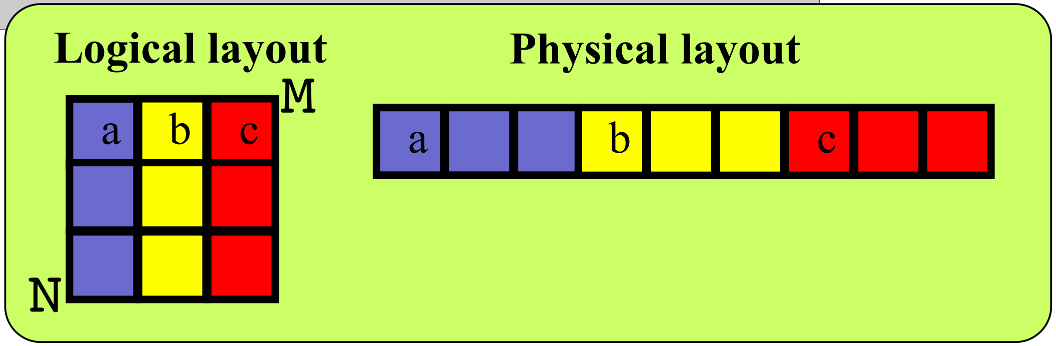
```
MPI_Type_free(newtype)
newtype   the type to free
```

- MPI datatypes cannot be used for defining variables.

Example: sending row in Fortran

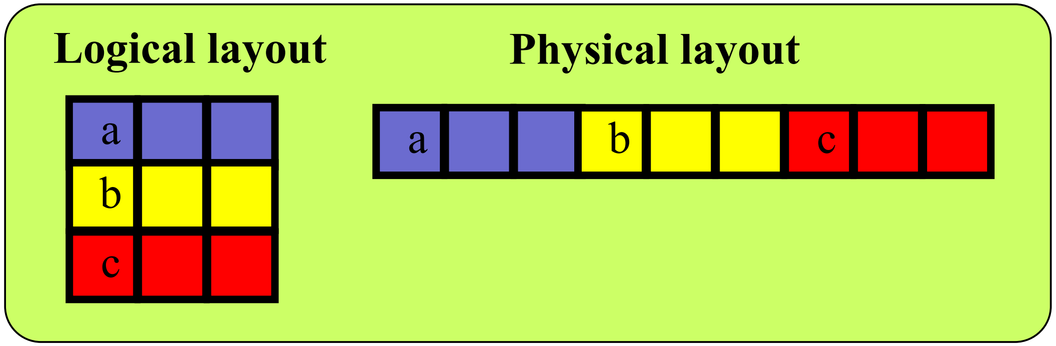
```
INTEGER, PARAMETER :: N=8, M=6
REAL, DIMENSION(N,M) :: A
INTEGER :: rowtype

! Create a derived type
CALL MPI_TYPE_VECTOR(M, 1, N, MPI_REAL, rowtype, ierr)
CALL MPI_TYPE_COMMIT(rowtype, ierr)

! Send a row
CALL MPI_SEND(A, 1, rowtype, dest, tag, comm, ierr)

! Free the type after it is not needed
CALL MPI_TYPE_FREE(rowtype, ierr)
```

number of elements
stride



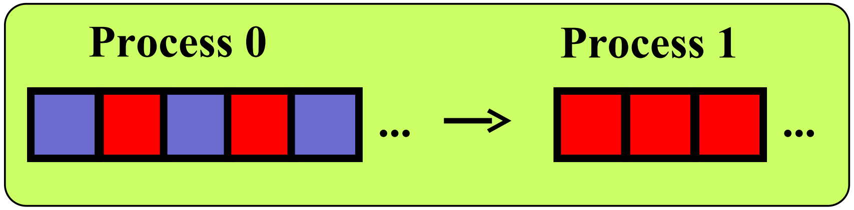
Example: sending column in C

```
int n=8, m=6;
float A[n,m];
MPI_Datatype columntype;

//Create a derived type
MPI_Type_Vector(n, 1, m, MPI_FLOAT, &columntype);
MPI_Type_Commit(&columntype);

// Send a column
MPI_Send(A, 1, columntype, dest, tag, comm);

//Free the type after it is not needed
MPI_Type_Free(&columntype);
```



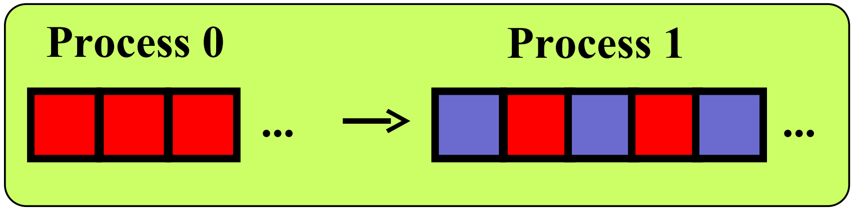
Data-types in communication

- Derived datatypes can be used both in
  - point-to-point communication
  - collective communication
- During the communication, the datatype tells MPI system where to take the data when sending or where to put data when receiving
  - a non-contiguous data in sending process can be received as contiguous or vice versa




```
if (myid = 0)
   MPI_Type_Vector(n, 1, 2, MPI_FLOAT, &newtype)
   ...
   MPI_Send(A, 1, newtype, 1, …)
else
   MPI_Recv(B, n, MPI_FLOAT,0, ...)
```

or




```
if (myid = 0)
   MPI_Send(A, n, MPI_FLOAT, 1, …)
else
   MPI_Type_Vector(n, 1, 2, MPI_FLOAT, &newtype)
   ...
   MPI_Recv(B, 1, newtype,0, ...)
```

MPI Type vector call

We need to tell MPI how the data is laid out

MPI_Type_vector(count, blocklen, stride, basetype, newtype) will create a new datatype, which consists of count instances of blocklen times basetype, with a space of stride in between.

stride = 4
count = 5
blocklen = 1

Before a new type can be used it has to be committed with MPI_Type_commit(MPI_Datatype* newtype)

```
MPI_Datatype newtype;
MPI_Type_vector(N, blocklen, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(m, 1, newtype, peer, tag, comm);
```

Exercise: DataTypes

1. MPI_Type_Vector (userdefined_types1)
2. MPI_Type_Indexed (userdefined_types2)
3. MPI_Type_Create_Subarray (userdefined_types3)

README explains the exercise:

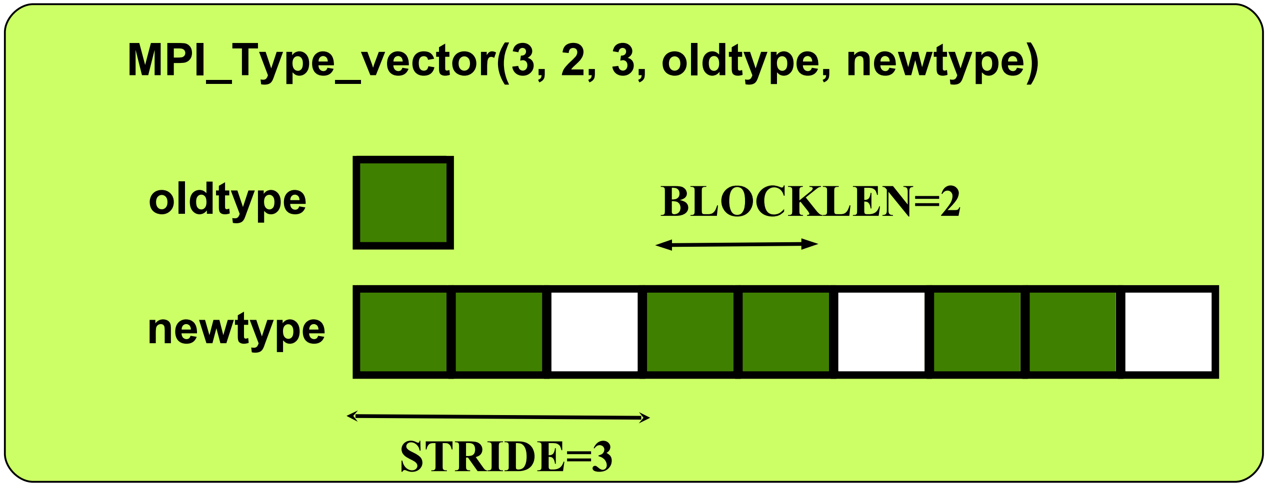
- compile and run the code and read the source files
- make small changes to the example files to transfer different array blocks

Next slides explain the three methods and can be used to make the exercise.

Datatype constructors: MPI_TYPE_VECTOR

- Creates a new type from equally spaced identical blocks.

```
MPI_Type_vector(count, blocklen, stride, oldtype, newtype)
count    number of blocks
blocklen number of elements in each block
stride   displacement between the blocks in extent of oldtype
```



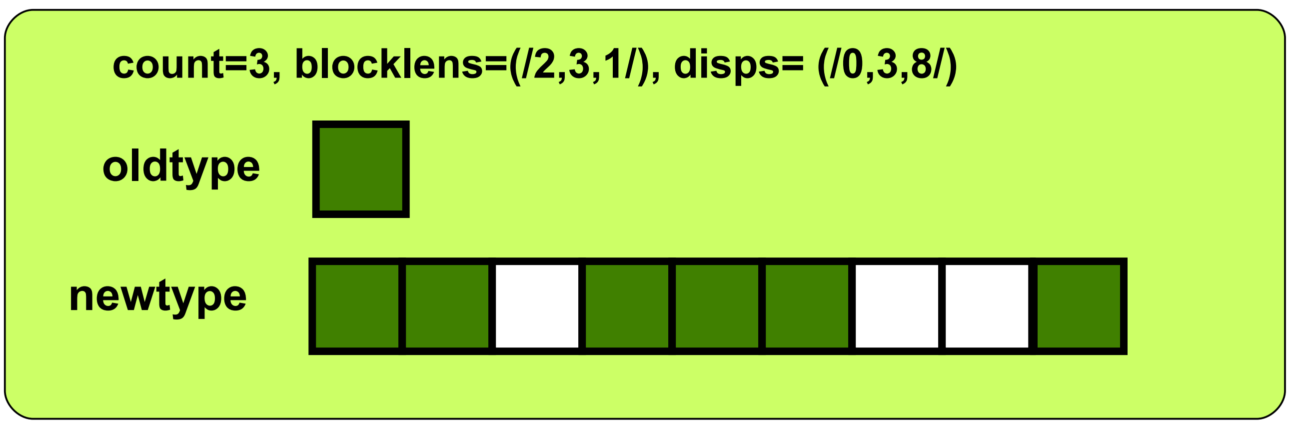
- **`MPI_Type_create_hvector`**
  - like **`MPI_Type_vector`**, but stride is in bytes

Datatype constructors: MPI_TYPE_INDEXED

- Creates a new type from blocks comprising identical elements. The size and displacements of the blocks can vary

  `MPI_Type_indexed(count, blocklens, displs, oldtype, newtype)`

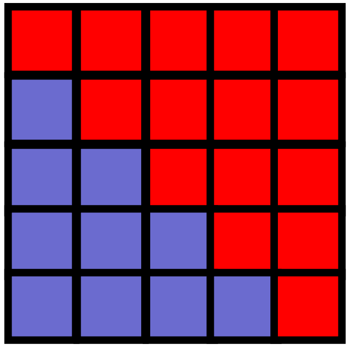
count	number of blocks
blocklens	lengths of the blocks (array)
displs	displacements (array) in extent of oldtypes



- **`MPI_Type_create_hindexed`**
  - like **`MPI_Type_indexed`** but displacements are in bytes

Example: send an upper triangular matrix

```
/* Upper triangular matrix */
double a[100][100];
int  disp[100], blocklen[100], int i;
MPI_Datatype upper;
/* compute start and size of rows */
for (i=0;i<100;i++)
  {
    disp[i]=100*i+i;
    blocklen[i]=100-i;
  }
/* create a datatype for upper triangular matrix */
MPI_Type_indexed(100,blocklen,disp,MPI_DOUBLE,&upper);
MPI_Type_commit(&upper);
/* ... send it ... */
MPI_Send(a,1,upper,dest, tag, MPI_COMM_WORLD);
MPI_Type_free(&upper);
```



Datatype constructors: MPI_TYPE_CREATE_SUBARRAY

- Creates a type describing a n-dimensional subarray of within n-dimensional array

```
MPI_Type_create_subarray(ndims, sizes, subsizes, offsets,
                         order, oldtype, newtype)
```

ndims	number of array dimensions
sizes	number of array elements in each dimension (array)
subsizes	number of subarray elements in each dimension (array)
offsets	starting point of subarray in each dimension (array)
order	storage order of the array. Either MPI_ORDER_C or MPI_ORDER_FORTRAN

Example

```
int array_size[2]={5,5};
int subarray_size[2]={2,2};
int subarray_start[2]={1,1};
MPI_Datatype subtype;
double **array

for(i=0;i<array_size[0];i++)
  for(j=0;j<array_size[1];j++)
    array[i][j]=rank;

MPI_Type_create_subarray(2,array_size,subarray_size,subarray_start,
                         MPI_ORDER_C,MPI_DOUBLE,&subtype);
MPI_Type_commit(&subtype);

if(rank==0)
   MPI_Recv(array[0],1,subtype,1,123,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
if (rank==1)
   MPI_Send(array[0],1,subtype,0,123,MPI_COMM_WORLD);
```

```
Rank 0: original array
   0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0
Rank 0: array after receive
   0.0    0.0    0.0    0.0    0.0
   0.0    1.0    1.0    0.0    0.0
   0.0    1.0    1.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0
```

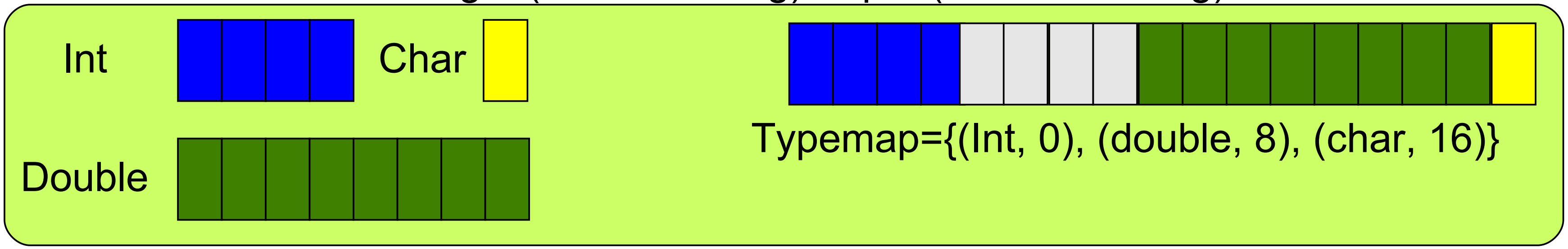
Derived Datatypes Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - basic datatype at displacement

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

TypeMap

- A datatype is defined by a typemap
  - pairs of basic types and displacements (in bytes)
  - typemap={(type 0, disp 0),...(type n-1, disp n-1)}
- E.g. MPI_INT={(int,0)}
- Type signature={type 0, type 1,...}
  - The type signature is the list of types in the typemap
  - Gives size of each element
  - Tells MPI how to interpret the bits it sends and receives
- Displacements
  - Tells MPI where to get (when sending) or put (when receiving)

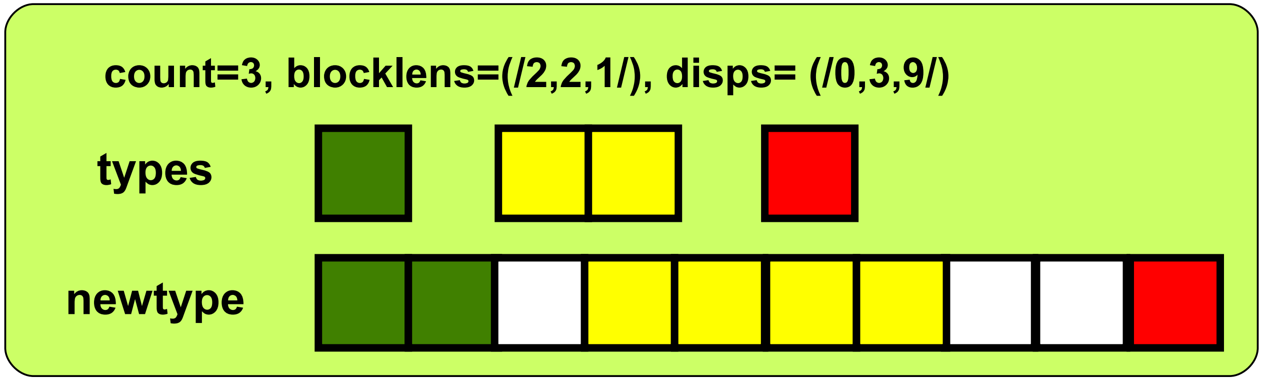


Datatype constructors: MPI_TYPE_CREATE_STRUCT

- Most general type constructor, creates a new type from heterogeneous blocks
  - E.g. Fortran 77 common blocks, Fortran 9x and C structures.

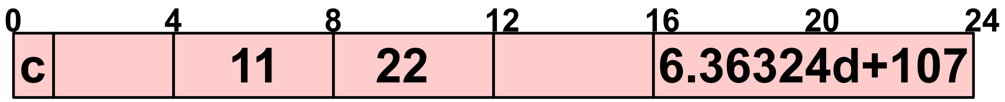
```
MPI_Type_create_struct(count, blocklens, displs, types, newtype)
```

count	number of blocks
blocklens	lengths of blocks (array)
displs	displacements of blocks in bytes (array)
types	types of blocks (array)



Example: sending a C struct

```
/* Structure for particles */
struct PartStruct {
   char class;  /* particle class */
   double d[6]; /* particle coords */
   char b[7];   /* additional info */ };

struct PartStruct particle[1000];
MPI_Datatype Particletype;

MPI_Datatype type[3]={MPI_CHAR, MPI_DOUBLE, MPI_CHAR};
int blocklen[3]={1,6,7};
/* MPI_Aint: an integer type that can  hold an arbitrary address.
Double  word alignment assumed. */
MPI_Aint disp[3]={0, sizeof(double), 7*sizeof(double)};
a
MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);

MPI_Send(particle, 1000, Particletype, dest,tag, MPI_COMM_WORLD);
MPI_Type_free(&Particletype);
```



Example:

derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g.,
structures,
common blocks,
subarrays,
some variables in the memory

Data Layout and the Describing Datatype Handle

```
struct buff_layout {
    int       i_val[3];
    double    d_val[5];
 }  buffer;
```

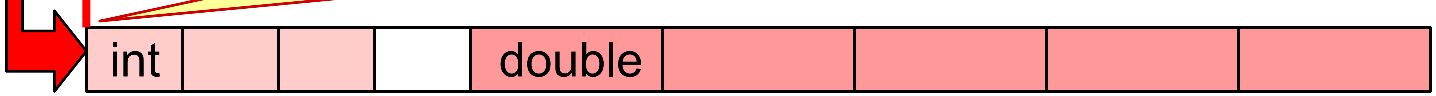
compiler

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=…;

MPI_Type_struct(2,  array_of_blocklengths,
      array_of_displacements, array_of_types,
      &buff_datatype);

MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, …)
```

&buffer = the start address of the data

the datatype handle describes the data layout

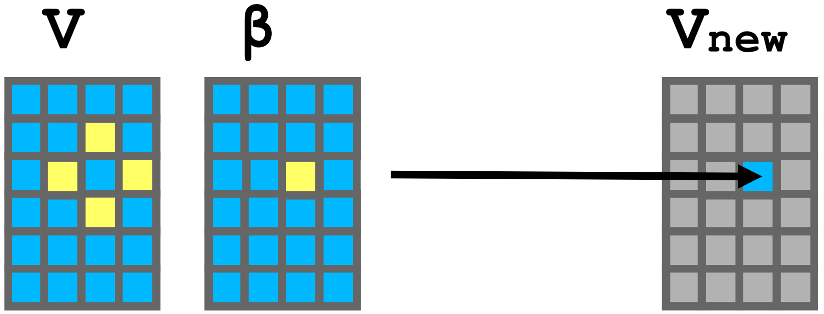


Example: MPI_Cart/jacobi.f90

- Jacobi solver for the Poisson equation $\nabla^2 V = f$: Iteratively update the value of a 2D array V

$$V_{new}(i,j)=0.25*[V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1)-\Delta^2 f(i,j)]$$

See `jacobi.f90`

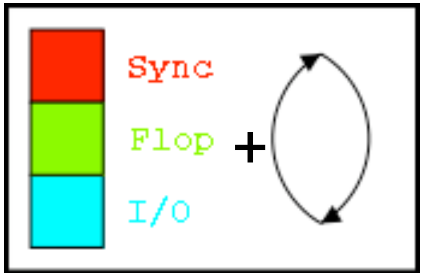


Load imbalance

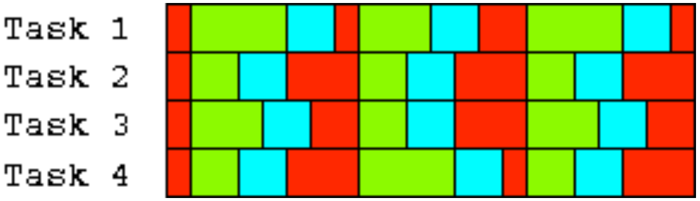
- Byproduct of data decomposition.
- Must be addressed by the decomposition step.

Load Balancing example

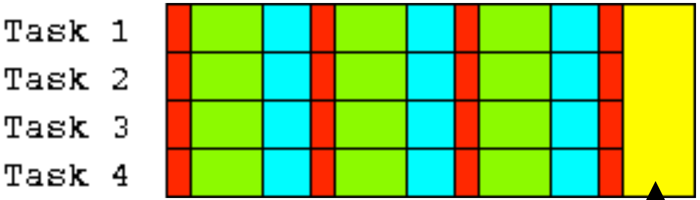
The Universal Parallel Science App



Unbalanced:



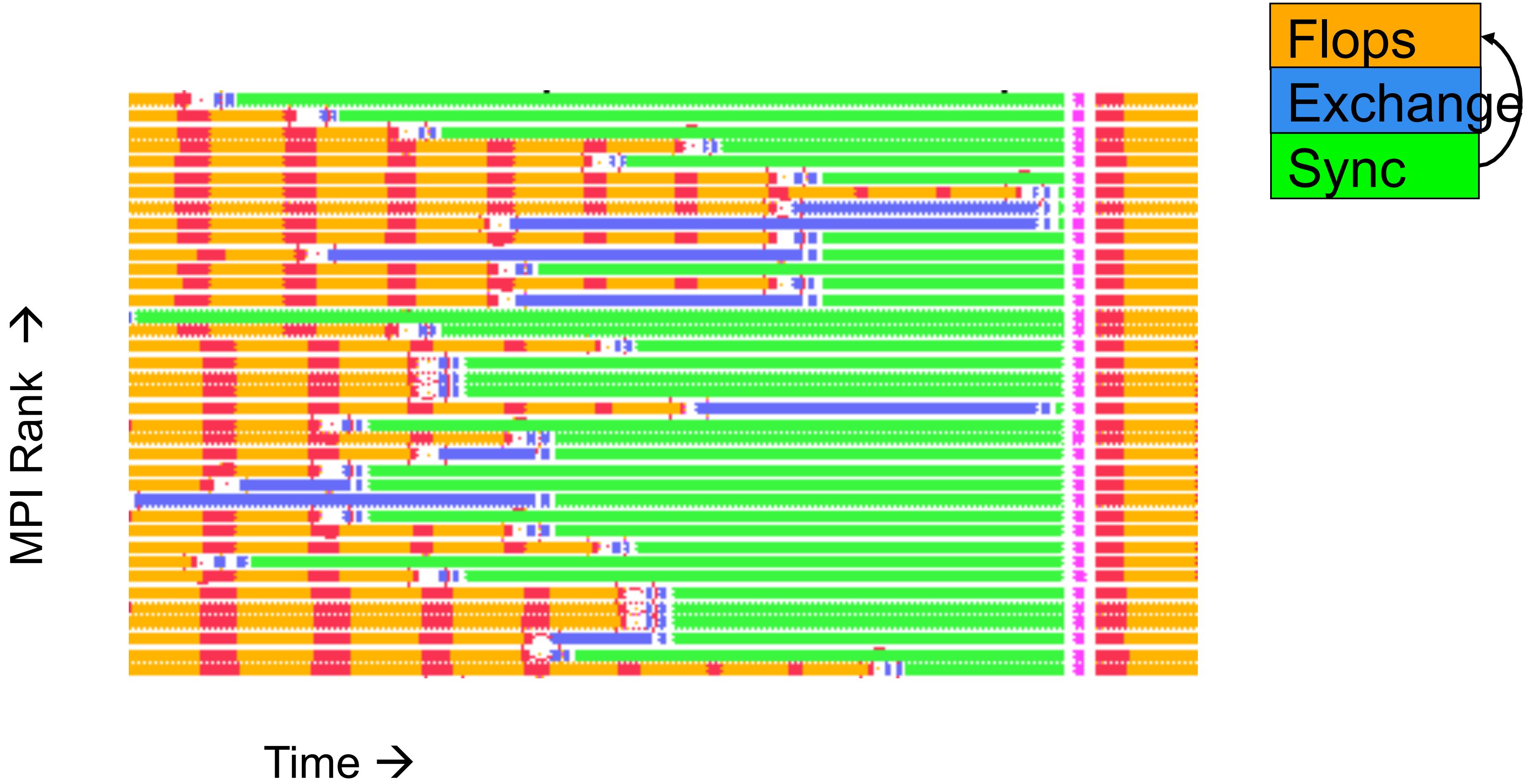
Balanced:



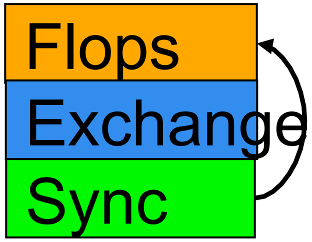
Time saved by load balance

~All apps come down to the same basic pattern. Ok, Maybe there is no I/IO.

Load Balancing example



Dynamic load balancing



Communication pattern