

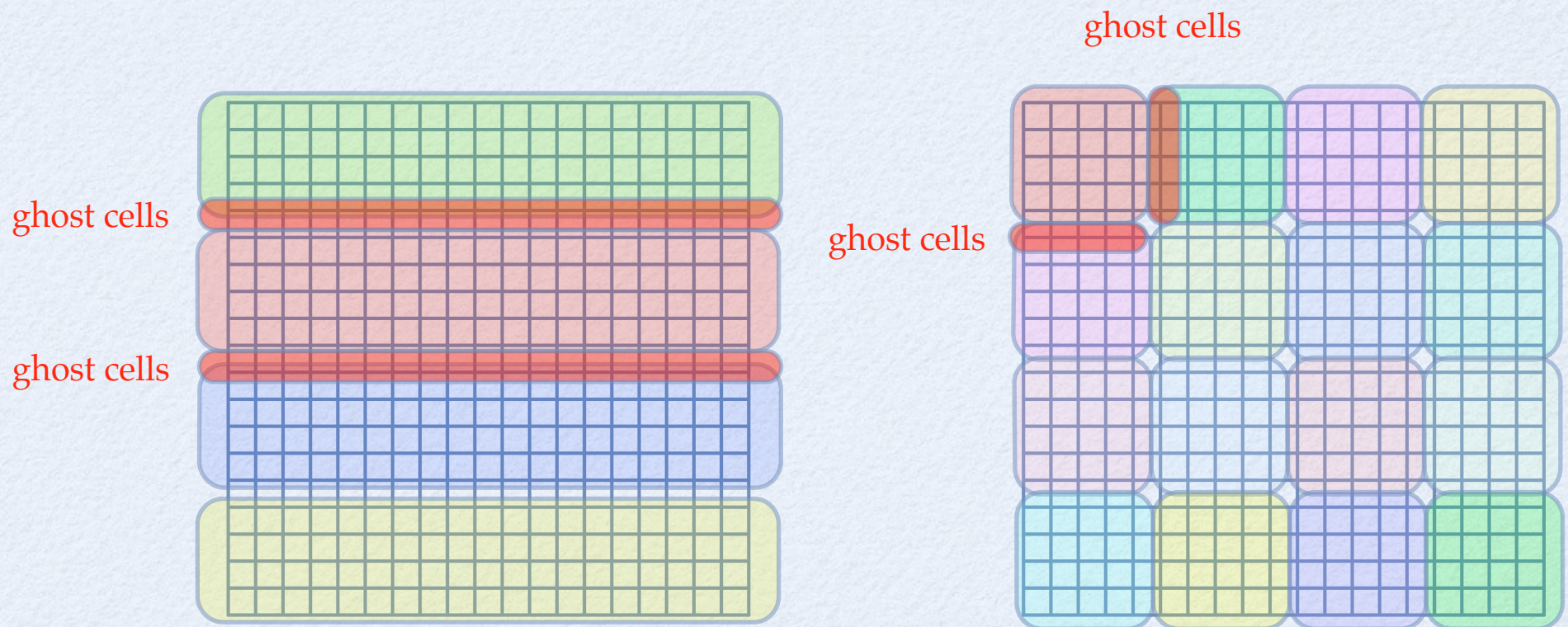
**HPCSE II**

**Advanced MPI**



# Recall domain decomposition and ghost cells

- How do we best exchange boundary values with the neighboring ranks?
  - In 1D it was just a single number and was easy
  - Sometimes we might be lucky and they could be contiguous arrays
- What shall we do in the general case?
  - pack them into buffers?
  - or just describe to MPI where they are in memory?





# Recall sending the parameters

- We had three options, none was ideal
  - three individual broadcasts: wasteful since three communications
  - packing it into a buffer: wasteful since it involves copying
  - sending the struct bitwise: dangerous since it assumes homogeneity
- What we want to do here and for the ghost cells is to send non-contiguous or heterogeneous data without copying.



# Broadcast

- MPI provides a collective broadcast operation

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,  
              MPI_Comm comm )  
// broadcast the data from the root rank to all others
```

- We can use this to broadcast the data

```
// and then broadcast the parameters to the other ranks  
MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
MPI_Bcast(&nsteps, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- This is inefficient since we use three broadcasts.
- We will later pack all parameters into one buffer and broadcast that buffer.

# Sending it bitwise

- The dangerous solution: pack it all into a struct and send it bitwise
- This assumes a homogeneous machine with identical integer and floating point formats.

```
// define a struct for the parameters
struct parms {
    double a;           // lower bound of integration
    double b;           // upper bound of integration
    int nsteps; // number of subintervals for integration
};

parms p;

// read the parameters on the master rank
if (rank==0);
    std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms as bytes - warning, not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
```



# Packing data into a buffer

- Pack the input data, broadcast it and unpack

```
// create a buffer and pack the values.
// first get the size for the buffer and allocate a buffer
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &size_double);
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];

// pack the values into the buffer on the master
if (rank==0) {
    int pos=0;
    MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
    MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
    MPI_Pack(&nsteps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
    assert ( pos <= buffer_size );
}

// broadcast the buffer
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

// and unpack on the receiving side
int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &nsteps, 1, MPI_INT, MPI_COMM_WORLD);
assert ( pos <= buffer_size );

// and finally delete the buffer
delete[] buffer;
```

# Recall sending the parameters

- We had three options, none was ideal
  - three individual broadcasts: wasteful since three communications
  - packing it into a buffer: wasteful since it involves copying
  - sending the struct bitwise: dangerous since it assumes homogeneity
- What we want to do here and for the ghost cells is to send non-contiguous or heterogeneous data without copying.
- The solution are MPI datatypes: describe your data layout to MPI and MPI uses that information in the communication.

```
struct parms {  
    double a;  
    double b;  
    int nsteps;  
};
```

type	count	offset
MPI_DOUBLE	2	0
MPI_INT	1	16



# Building an MPI data type

- The most general function is `MPI_Type_create_struct`, taking numbers, offsets and types

```
// define a struct for the parameters
struct parms {
    double a;           // lower bound of integration
    double b;           // upper bound of integration
    int nsteps; // number of subintervals for integration
};

// describe this struct through sizes, offsets and types
// and create an MPI data type
// still dangerous since it assumes that we know any potential padding
MPI_Datatype parms_t;
int          blocklens[2] = {2,1};
MPI_Aint     offsets[2]   = {0,2*sizeof(double)};
MPI_Datatype types[2]    = {MPI_DOUBLE, MPI_INT};
MPI_Type_create_struct(2, blocklens, offsets, types,&parms_t);
MPI_Type_commit(&parms_t); // finish building the type

parms p;

// read the parameters on the master rank
if (rank==0);
    std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms now using our type
MPI_Bcast(&p, 1, parms_t, 0, MPI_COMM_WORLD);

// and now free the type
MPI_Type_free(&parms_t);
```

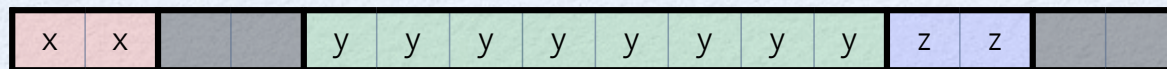
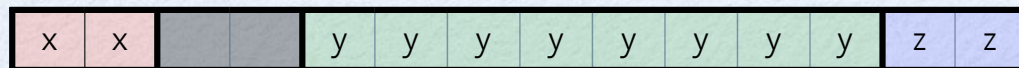
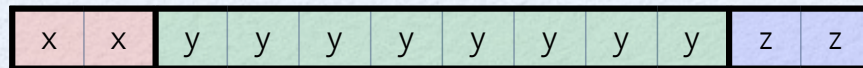


# Alignment and padding

- This code was dangerous since we assumed that we know how the compiler lays out a struct in memory.

```
struct parms {  
    short x;  
    double y;  
    short z;  
};
```

- We might be wrong due to padding and alignment. Consider the following three examples of how this could be stored in memory



↑  
Padding to ensure alignment of the next variable

↑

# Safer way of using MPI\_Type\_create\_struct

- To get the right offsets and size we take actual addresses
  - use MPI\_Get\_address to convert pointers to integers
  - specify lower bound and upper bound of the struct

```
// define a struct for the parameters
struct parms {
    double a;           // lower bound of integration
    double b;           // upper bound of integration
    int nsteps; // number of subintervals for integration
};

parms p;

// describe the struct through sizes, offsets and types
// the safe way getting addresses

MPI_Aint p_lb, p_a, p_nsteps, p_ub;
MPI_Get_address(&p, &p_lb); // start of the struct is the lower bound
MPI_Get_address(&p.a, &p_a); // address of the first double
MPI_Get_address(&p.nsteps, &p_nsteps); // address of the integer
MPI_Get_address(&p+1, &p_ub); // start of the next struct is the upper bound

int blocklens[] = {0, 2, 1, 0};
MPI_Datatype types[] = {MPI_LB, MPI_DOUBLE, MPI_INT, MPI_UB};
MPI_Aint offsets[] = {0, p_a-p_lb, p_nsteps-p_lb, p_ub-p_lb};

MPI_Datatype parms_t;
MPI_Type_create_struct(4, blocklens, offsets, types, &parms_t);
MPI_Type_commit(&parms_t);
```



# MPI\_Type\_create\_struct

- The declaration of the functions used in the previous examples

```
int MPI_Type_create_struct(int count, int blocklengths[], MPI_Aint offsets[],
                          MPI_Datatype types[], MPI_Datatype *newtype)
// builds an MPI data type for a data type for a general data structure given by
// types, counts (blocklengths) and their offsets relative to the start of the data structure

int MPI_Get_address(void *location, MPI_Aint *address)
// converts a pointer to the integer type used internally by MPI to store pointers

int MPI_Type_commit(MPI_Datatype *datatype)
// commits the data type: finished building it. It can now be used.

int MPI_Type_free(MPI_Datatype *datatype)
// frees the data type, releasing any allocated memory
```

- We can use MPI\_Type\_create\_struct to send the contents of linked lists
  - we view the whole memory as a huge struct from which we send some select data
  - thus give absolute addresses as offsets
  - pass **MPI\_BOTTOM** as the buffer pointer in communication to indicate that the type uses absolute addresses

# Receiving a list into a vector

```
if(num==0) {
    // receive data into a vector and print it
    std::vector<int> data(10);
    MPI_Status status;
    MPI_Recv(&data[0], 10, MPI_INT, 1, 42, MPI_COMM_WORLD, &status);
    for (int i=0; i < data.size(); ++i)
        std::cout << data[i] << "\n";
}
else {
    // fill a list with the numbers 0-9 and send it
    std::list<int> data;
    for (int i=0; i<10; ++i)
        data.push_back(i);

    std::vector<MPI_Datatype> types(10,MPI_INT);
    std::vector<int>          blocklens(10,1);
    std::vector<MPI_Aint>    offsets;

    for (int& x : data) {
        MPI_Aint address;
        MPI_Get_address(&x, &address); // use absolute addresses
        offsets.push_back(address);
    }

    MPI_Datatype list_type;
    MPI_Type_create_struct(10, &(blocklens[0]), &offsets[0], &types[0] ,&list_type);
    MPI_Type_commit(&list_type);

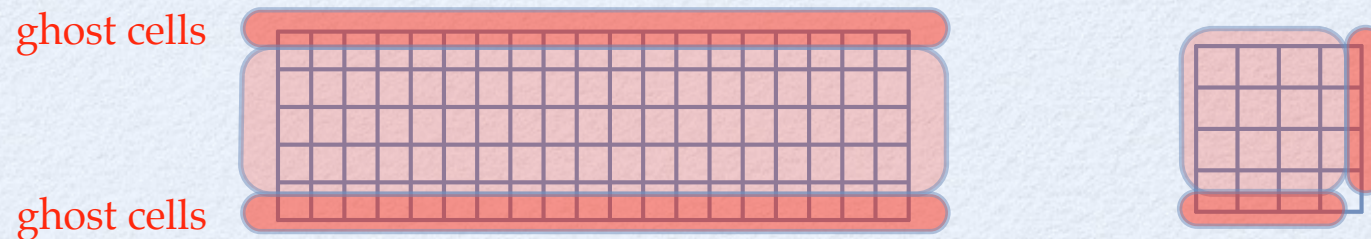
    MPI_Send(MPI_BOTTOM, 1, list_type, 0, 42, MPI_COMM_WORLD);

    MPI_Type_free(&list_type);
}
```



# MPI data types for ghost cells

- The ghost cells in a 2D array or column and rows in a matrix can be described as strided vectors



```
int MPI_Type_contiguous(int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)
// build an MPI datatype for a contiguous array

int MPI_Type_vector(int count, int blocklength, int stride,
                   MPI_Datatype old_type, MPI_Datatype *newtype_p)
// build an MPI datatype for a vector array of blocklength contiguous entries that are
// spaced at a given stride. stride is like the leading dimension in BLAS and specifies
// the distance between blocks

int MPI_Type_create_hvector(int count, int blocklen, MPI_Aint stride,
                           MPI_Datatype old_type, MPI_Datatype *newtype_p)
// like MPI_Type_vector but now the stride is given in bytes
```



# Row and column data types

- We can use this to create data types for rows and columns of a matrix, and similarly for slices of an array

```
hpc12::matrix<double,hpc12::column_major> a(4,4);

MPI_Datatype row, col;

MPI_Type_contiguous(4, MPI_DOUBLE, &col);
MPI_Type_vector(4, 1, 4, MPI_DOUBLE, &row);

MPI_Type_commit(&row);
MPI_Type_commit(&col);

// use them
// ...
// and finally free them

MPI_Type_free(&row);
MPI_Type_free(&col);
```

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16



# Subarrays

- More general is the creation of subarrays, especially for boundary layers and ghost cells

```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[],
                             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
// build an MPI datatype for a subarray of a larger array:
//  ndims:    number of dimensions
//  sizes:    extent of the full array in each dimension
//  subsizes: extent of the subarray in each dimension
//  starts:   starting index of the subarray
//  order:    array storage order, can be either of MPI_ORDER_C or MPI_ORDER_FORTRAN
```

- Use it for the 2D diffusion equation parallelized using MPI



# Indexed data types

- Finally, we want to send just some elements of an array. For example, send some particles to a different cell list.

```
int MPI_Type_indexed(int count, int blocklens[], int indices[],
                    MPI_Datatype old_type, MPI_Datatype *newtype)
// build an MPI datatype selecting specific entries from a contiguous array. Starting a
// at each of the given indices a number of elements given in the corresponding entry
// of blocklens is chosen.

int MPI_Type_create_hindexed(int count, int blocklens[], MPI_Aint displacements[],
                            MPI_Datatype oldtype, MPI_Datatype *newtype)
// same as MPI_Type_indexed but now instead of indices the displacement in bytes from the
// start of the array is specified

int MPI_Type_create_indexed_block(int count, int blocklength, int array_of_displacements[],
                                  MPI_Datatype oldtype, MPI_Datatype *newtype)
// same as MPI_Type_indexed but with constant sized blocks
```



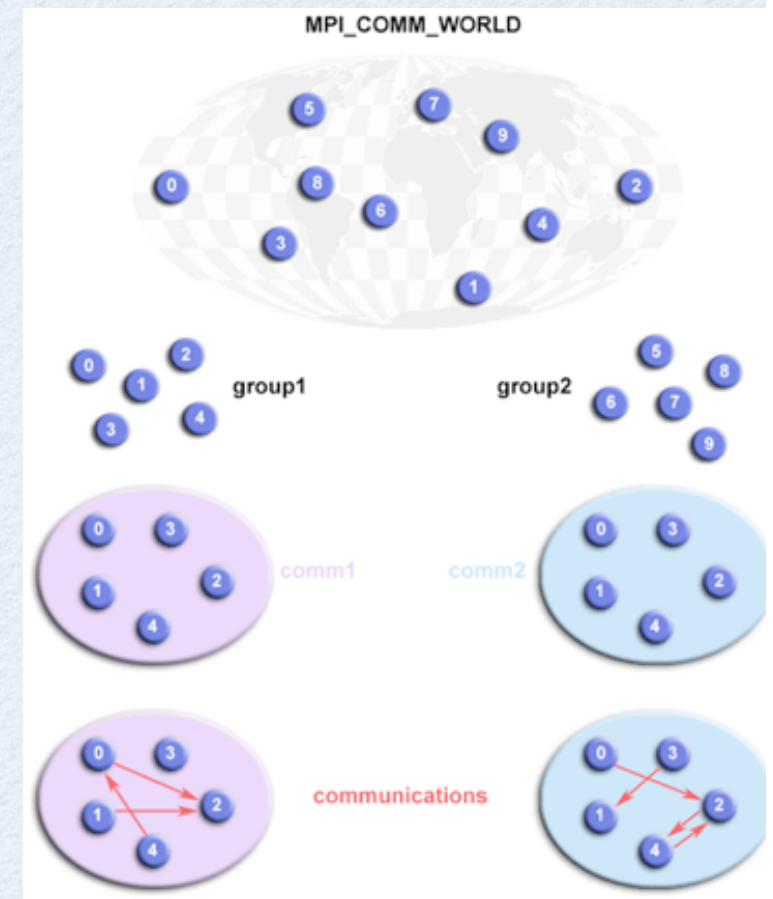
# Parallelizing your codes with MPI

- Let us now discuss how to parallelize the various codes you wrote so far and what MPI features to use:
  - Monte Carlo simulations
  - Partial differential equations
  - N-body codes with long range forces
  - N-body codes with short range forces
  - Linear algebra
- How would you do the following with MPI datatypes?
  - send some particles from one node to another? (hint: particles1.cpp)
  - send just the positions of some particles to another node? (hint: particles2.cpp)
  - send the positions of one particle to another node and receive them separately as x, y, and z coordinates?



# Groups and Communicators

- Imagine we want to split a computation into individual tasks that run on subsets of the ranks:
  - do multiple integrations at the same time
  - operate on rows or columns of a matrix
  - operate on slices of a 3D mesh
- We want to split the ranks into groups and build a new communicator for each group
- We can then do collective operations within a group instead of within all ranks





# Simpson using a communicator

- Simpson integration by MPI using a communicator that might be other than `MPI_COMM_WORLD`

```
double parallel_simpson(MPI_Comm comm, parms p)
{
    // get the rank and size for the current communicator
    int size;
    int rank;
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);

    // integrate just one part on each rank
    double delta = (p.b-p.a)/size;
    double result = simpson(func, p.a+rank*delta, p.a+(rank+1)*delta, p.nsteps/size);

    // collect the results to all ranks
    MPI_Allreduce(MPI_IN_PLACE, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
    return result;
}
```

# Three Simpson integrations at once

```
int main(int argc, char** argv)
{
    MPI_Init(&argc,&argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    // we want to do three integrals at once
    parms p[3];
    ...

    // split the ranks into three groups
    int which = rank % 3;
    MPI_Comm comm;
    MPI_Comm_split(MPI_COMM_WORLD, which, rank, &comm);

    // do the integration in each group
    double result = parallel_simpson(comm,p[which]);

    // only the master for each group prints
    int grouprank;
    MPI_Comm_rank(comm, &grouprank);
    if (grouprank==0)
        std::cout << "Integration " << which << " results in " << result << std::endl;

    // free the type and the new communicator
    MPI_Comm_free(&comm);

    MPI_Finalize();
    return 0;
}
```



# Creating and destroying communicators

- The most important functions for communicators

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

```
// compares two communicators to test if they are the same, i.e. they have the same ranks  
// in the same order
```

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
// duplicates a communicator.  
// this is a collective communication that needs to be called by all ranks.
```

```
int MPI_Comm_free(MPI_Comm *comm)
```

```
// frees a communicator
```

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

```
// splits a communicator into subcommunicators.  
// ranks with the same color are grouped together and sorted within each group by key.  
// this is a collective communication that needs to be called by all ranks.
```

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

```
// creates a new communicator based on group that is a subgroup of the ranks in comm.  
// this function allows more flexible creation of subcommunicators than MPI_Comm_split.  
// this is a collective communication that needs to be called by all ranks.
```



# Working with groups (1)

- There are **many** useful functions for group creation

```
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_size(MPI_Group group, int *size)
// are similar to the corresponding communicator functions

int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
// translates ranks between group: given a set of ranks1 in group1 it sets their ranks in group2
// in the array ranks2, or sets them to MPI_UNDEFINED if no correspondence exists

int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
// extracts the group from a communicator

int MPI_Group_free(MPI_Group *group)

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
// newgroup is the union, intersection, or difference of the given groups
```



# Working with groups (2)

- selectively choosing ranks

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
// create a newgroup containing only the given ranks of a group

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
// create a newgroup containing all except the given ranks of a group

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
// create a newgroup containing only the given ranges of ranks of a group

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
// create a newgroup containing all except the given ranges of ranks of a group
```

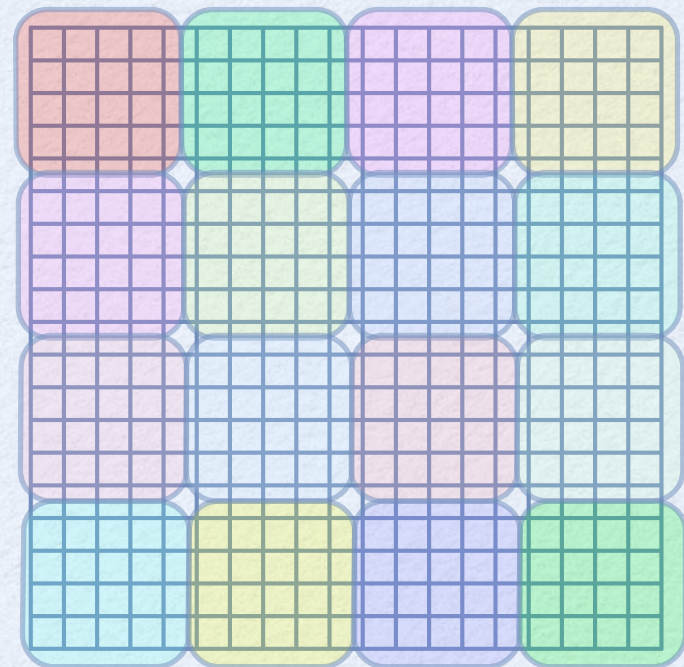
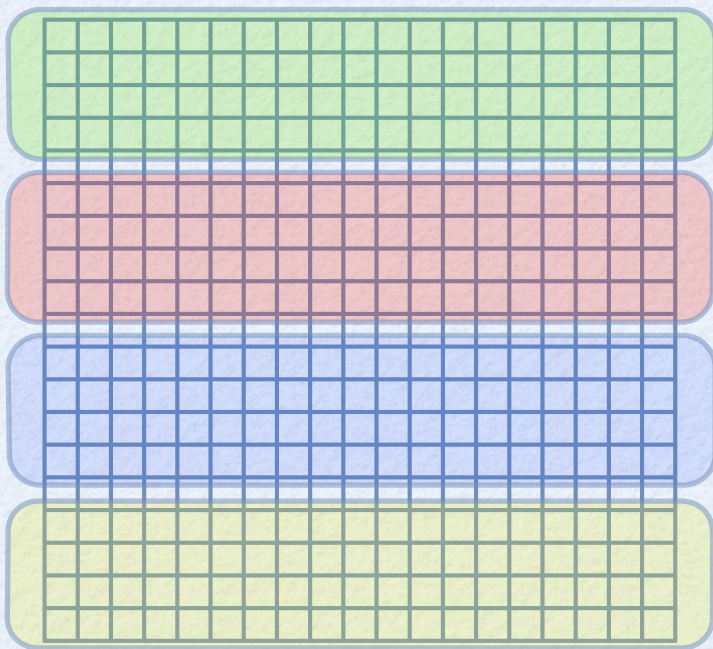
- ranges are given as triples (first, last, stride), and a range includes the ranks

$$\text{first, first+ stride, first+ 2 stride, ..., first+ } \left\lfloor \frac{\text{last} - \text{first}}{\text{stride}} \right\rfloor \text{stride}$$



# Finding the rank of the neighbor

- Easy in a one-dimensional layout
- Harder in two and more dimensions
- Even harder on irregular meshes



- MPI topologies are the solution to easily finding neighbors



# MPI topologies

- A (virtual) topology describes the “connectivity” of MPI processes in a communicator. There may be no relation between the physical network and the process topology.
- Two main types
  - **Cartesian topology:** each process is “connected” to its neighbors in a virtual grid. Nodes are labeled by cartesian indices, boundaries can be cyclic (periodic).
  - **Graph topology:** an arbitrary connection graph
- Topologies are essentially a simple graph library built into MPI



# Cartesian topologies: MPI\_Cart\_create

- To work with a regular mesh with row-major ordering we create a cartesian communicator

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,  
                  int reorder, MPI_Comm *comm_cart)
```

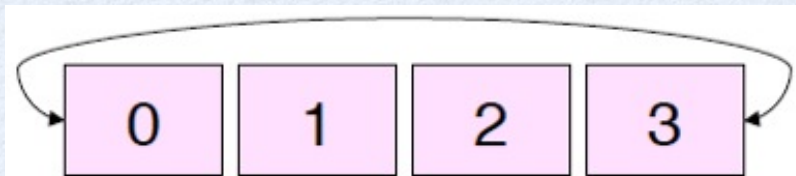
- **comm\_old**: the original communicator
  - **ndims**: number of dimensions
  - **dims**: integer array specifying the number of processes in each dimension
  - **periods**: integer array of boolean values whether the grid is periodic in that dimension
  - **reorder**: boolean flag whether the processes may be reordered
  - **comm\_cart**: a new cartesian grid communicator
- To get an automatic splitting into approximately equal counts in each dimension use

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)  
// fills in the dims array to be the best fit of arranging nnodes ranks to  
// form an ndims dimensional array
```



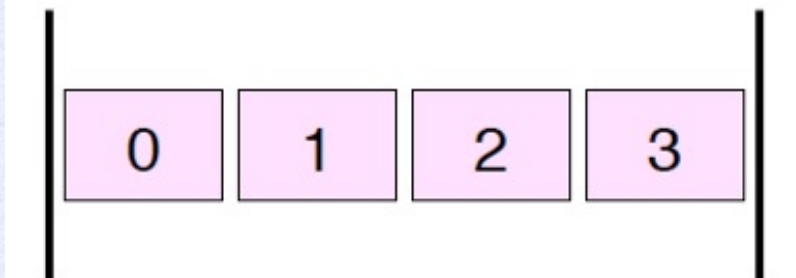
# Periodic boundary conditions

`int periods[] = {true}`



$left_0 = 3$   
 $right_3 = 0$

`int periods[] = {false}`



$left_0 = MPI\_PROC\_NULL$   
 $right_3 = MPI\_PROC\_NULL$

# Creating a cartesian communicator

```
int main(int argc, char** argv)
{
    // now initialize MPI and get information about the number of processes

    MPI_Init(&argc,&argv);

    int size;
    int rank;
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    int nums[3] = {0,0,0};
    int periodic[3] = {false, false, false};

    // split the nodes automatically
    MPI_Dims_create(size, 3, nums);

    if (rank==0)
        std::cout << "We create a " << nums[0] << "x" << nums[1] << "x" << nums[2] << " arrangement.\n";

    // now everyone creates a a cartesian topology
    MPI_Comm cart_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 3, nums, periodic, true, &cart_comm);

    MPI_Comm_free(&cart_comm);
    MPI_Finalize();
}
```



# The most important one: MPI\_Cart\_shift

- The neighbors are obtained by

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displacement, int *source, int *dest)
// gives the ranks shifted in the dimension given by direction by a certain displacement, where the
// sign of displacement indicates the direction.
// It returns both the source rank from which the current rank can be reached by that shift
// and the dest rank that is reached from the current rank by that shift.
```

- Example in 3D:

```
int left, right, bottom, top, front, back, newrank;

MPI_Comm_rank(cart_comm, &newrank);

MPI_Cart_shift(cart_comm, 0, 1, &left, &right);
MPI_Cart_shift(cart_comm, 1, 1, &bottom, &top);
MPI_Cart_shift(cart_comm, 2, 1, &front, &back);

std::cout << "Rank " << rank << " has new rank " << newrank << " and neighbors "
  << left << ", " << right << ", "
  << top << ", " << bottom << ", "
  << front << ", " << back << std::endl;
```



# Functions for cartesian topologies

- Get number of dimensions

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

- Get the cartesian topology information

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)  
// retrieves information about the cartesian topology associated with a communicator.  
// The arrays are allocated with maxdims dimensions. dims and periods are the numbers used  
// when creating the topology. coords are the dimensions of the current rank.
```

- Get the rank of a given coordinate

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

- Get the coordinates of a given rank

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

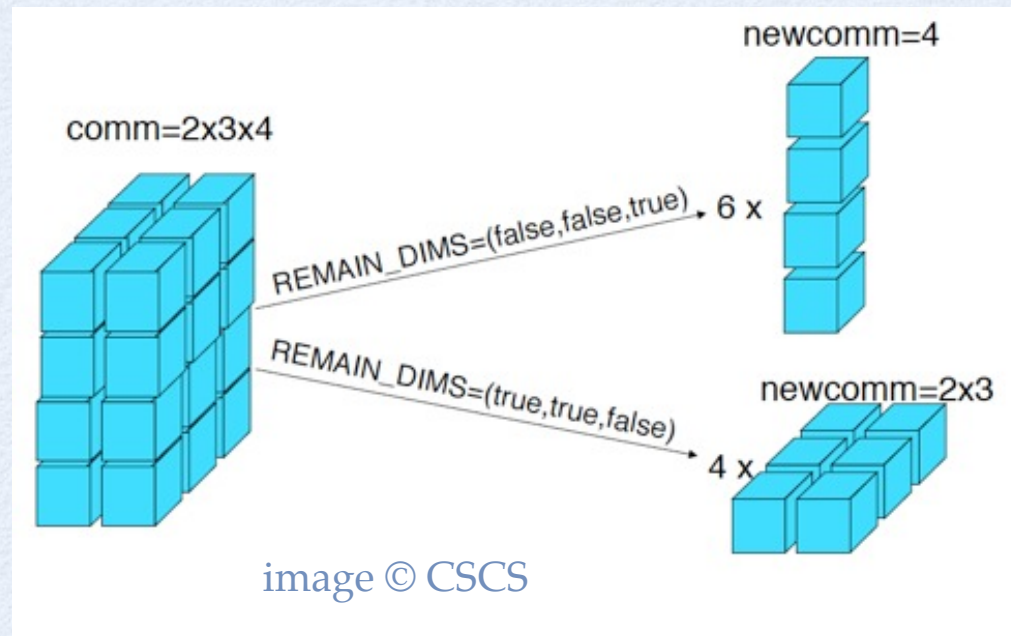


# Subgrids of cartesian topologies

- One can split the cartesian communicator into sub grid communicators for columns, rows, planes, ....

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *comm_new)
```

- The remain\_dims array specifies whether to keep the processes along a direction joined in a group (true) or split them (false)





# Graph topologies

- MPI contains another type of topology: a **graph topology**, which is not limited to a regular mesh.
  - Arbitrary number of neighbors for each rank
  - Useful for unstructured grids
- This is essentially a graph library. In C++ we can use nicer C++ graph libraries for the same functionality.