MPI 3.0 Neighbourhood Collectives

Advanced Parallel Programming

Overview

- Review of topologies in MPI

- MPI 3.0 includes new neighbourhood collective operations:
  - MPI_Neighbor_allgather[v]
  - MPI_Neighbor_alltoall[v|w]
- Example usage:
  - Halo-exchange can be done with a single MPI communication call

- Practical tomorrow:
  - Replace all point-to-point halo-exchange communication with a single neighbourhood collective in your MPP coursework code

Topology communicators (review 1)

- Regular n-dimensional grid or torus topology
  - MPI_CART_CREATE
- General graph topology
  - MPI_GRAPH_CREATE
    - All processes specify all edges in the graph (not scalable)
- General graph topology (distributed version)
  - MPI_DIST_GRAPH_CREATE_ADJACENT
    - All processes specify their incoming and outgoing neighbours
  - MPI_DIST_GRAPH_CREATE
    - Any process can specify any edge in the graph (too general?)

Topology communicators (review 2)

- Testing the topology type associated with a communicator
  - MPI_TOPO_TEST
- Finding the neighbours for a process
  - MPI_CART_SHIFT

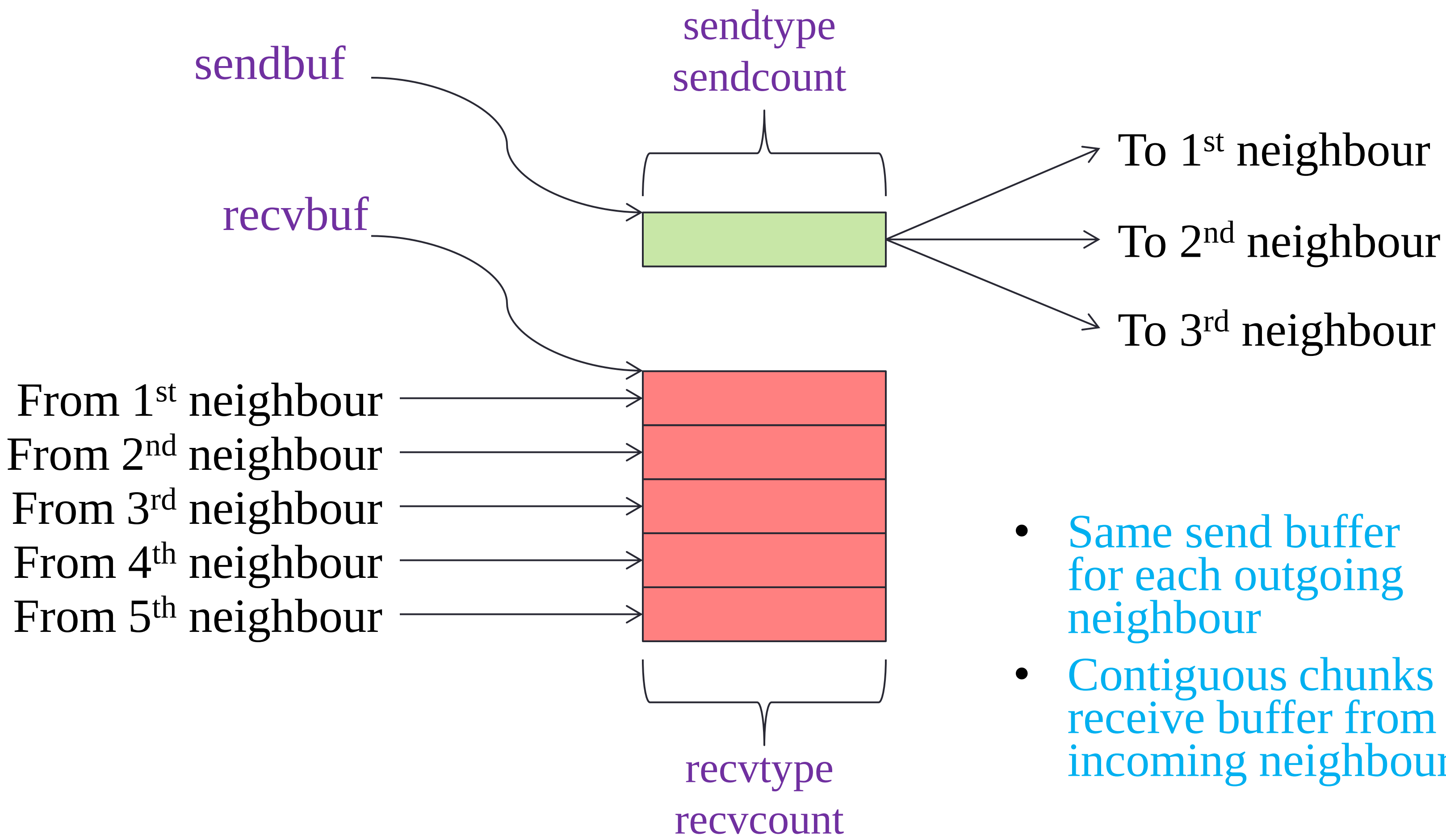
  - Find out how many neighbours there are:
    - MPI_GRAPH_NEIGHBORS_COUNT
  - Get the ranks of all neighbours:
    - MPI_GRAPH_NEIGHBORS

  - Find out how many neighbours there are:
    - MPI_DIST_GRAPH_NEIGHBORS_COUNT
  - Get the ranks of all neighbours:
    - MPI_DIST_GRAPH_NEIGHBORS

Neighbourhood collective operations

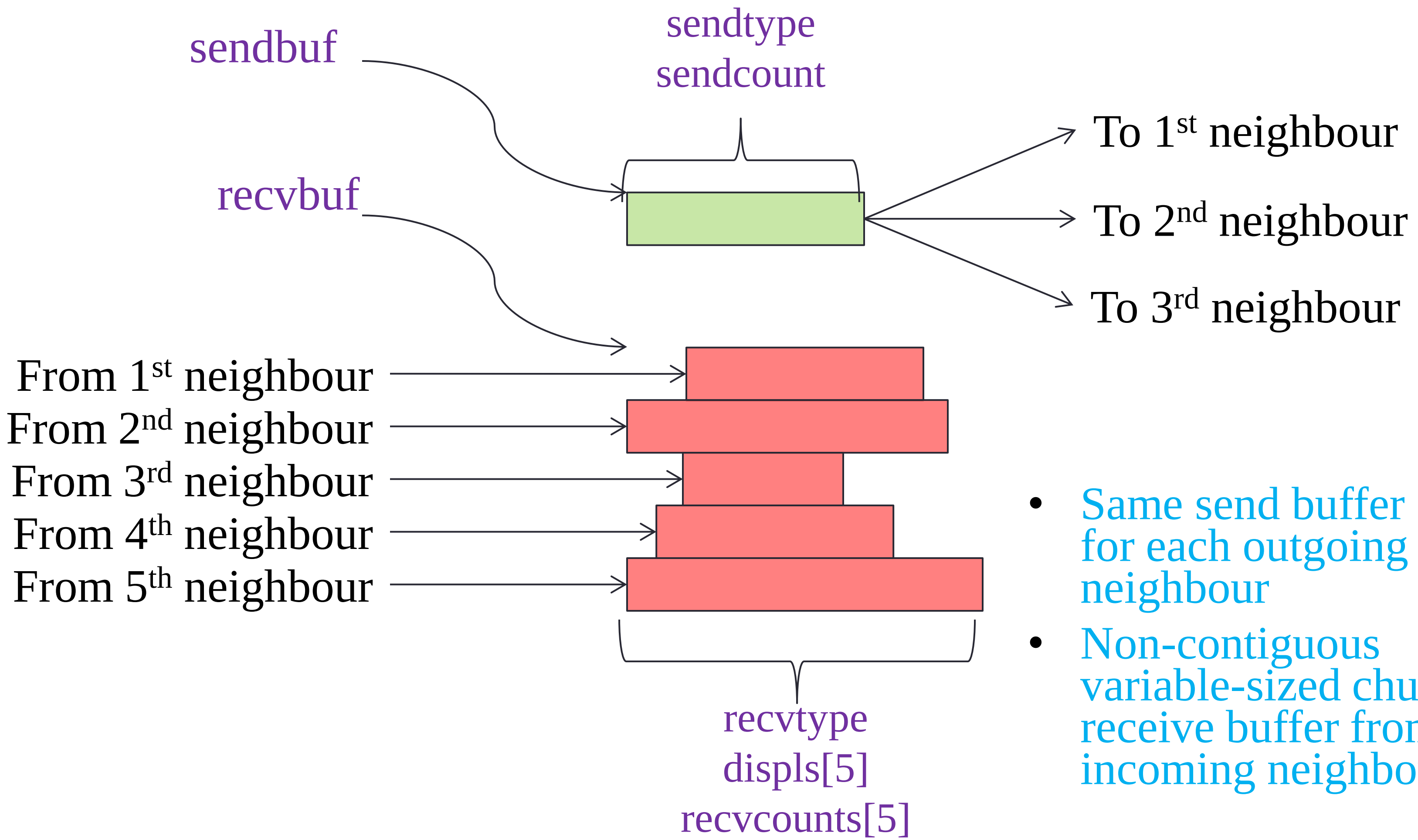
- See section 7.6 in MPI 3.0 for blocking functions
  - See section 7.7 in MPI 3.0 for non-blocking functions
  - See section 7.8 in MPI 3.0 for an example application
    - But beware of the mistake(s) in the example code!
- MPI_[N|In]eighbor_allgather[v]
  - Send one piece of data to all neighbours
  - Gather one piece of data from each neighbour
- MPI_[N|In]eighbor_alltoall[v|w]
  - Send different data to each neighbour
  - Receive different data from each neighbour
- Use-case: regular or irregular domain decomposition codes
  - Where the decomposition is static or changes infrequently
  - Because creating a topology communicator takes time

MPI_Neighbor_allgather



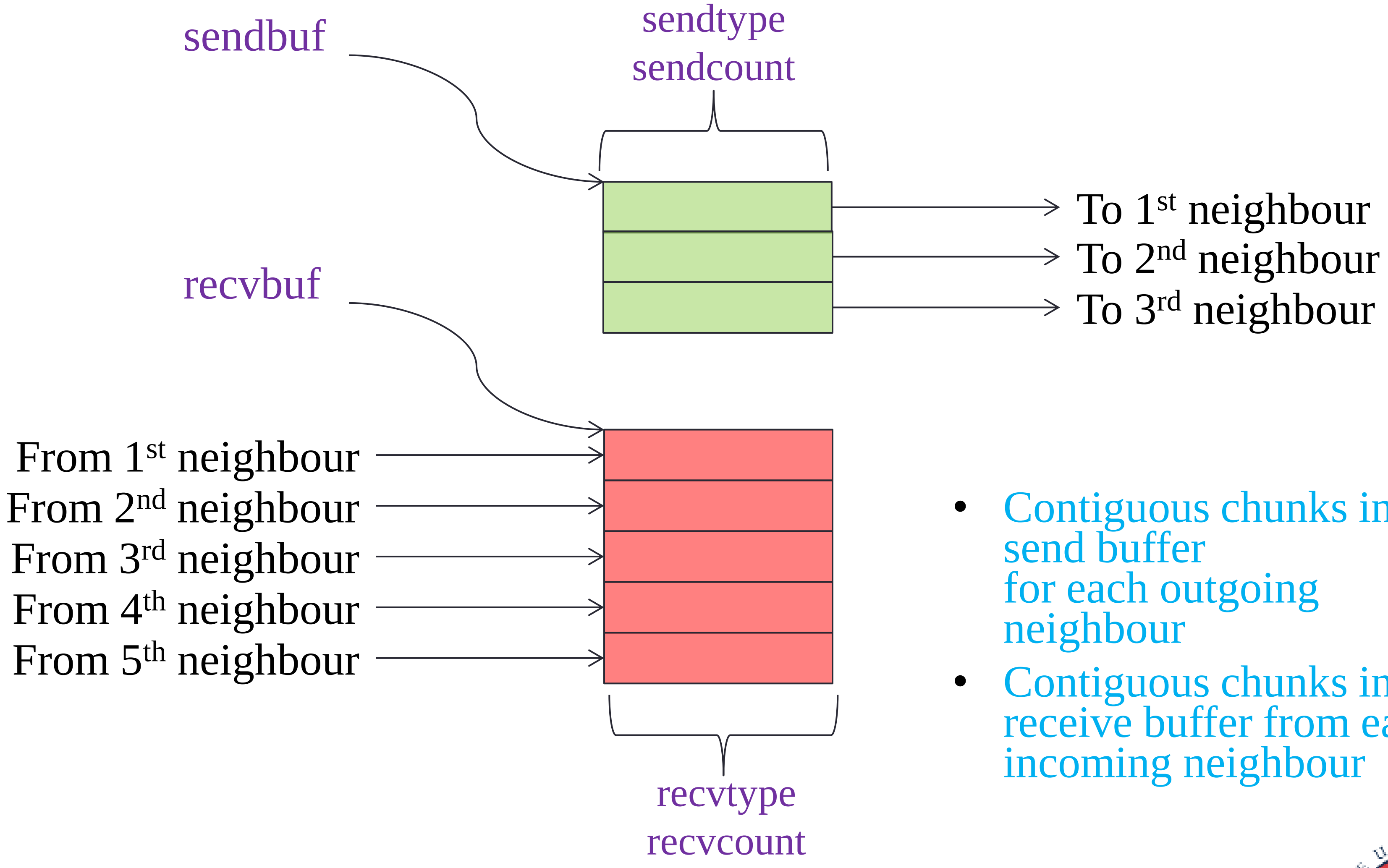
- Same send buffer for each outgoing neighbour
- Contiguous chunks in receive buffer from each incoming neighbour

MPI_Neighbor_allgatherv



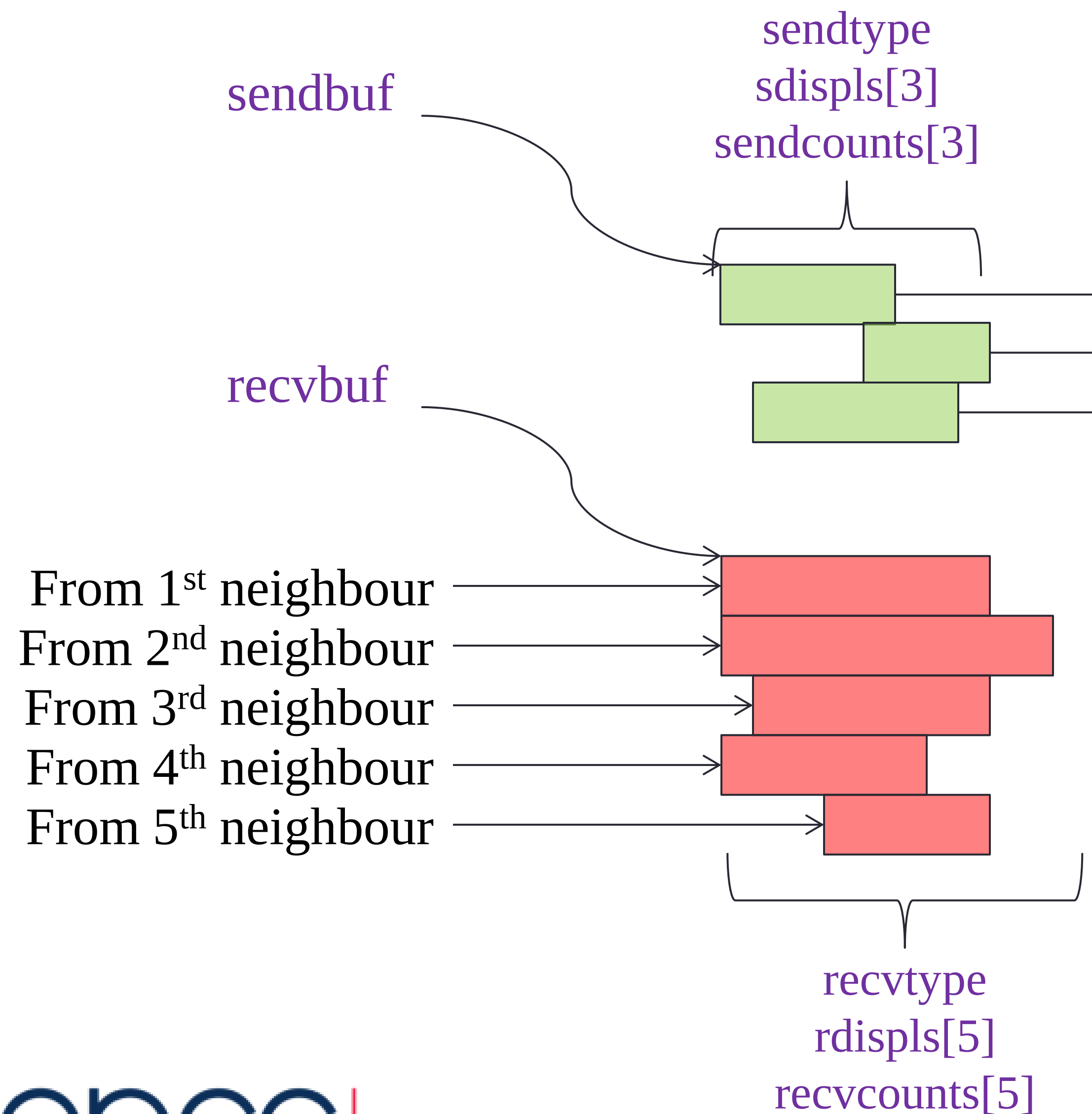
- Same send buffer for each outgoing neighbour
- Non-contiguous variable-sized chunks in receive buffer from each incoming neighbour

MPI_Neighbor_alltoall



- Contiguous chunks in send buffer for each outgoing neighbour
- Contiguous chunks in receive buffer from each incoming neighbour

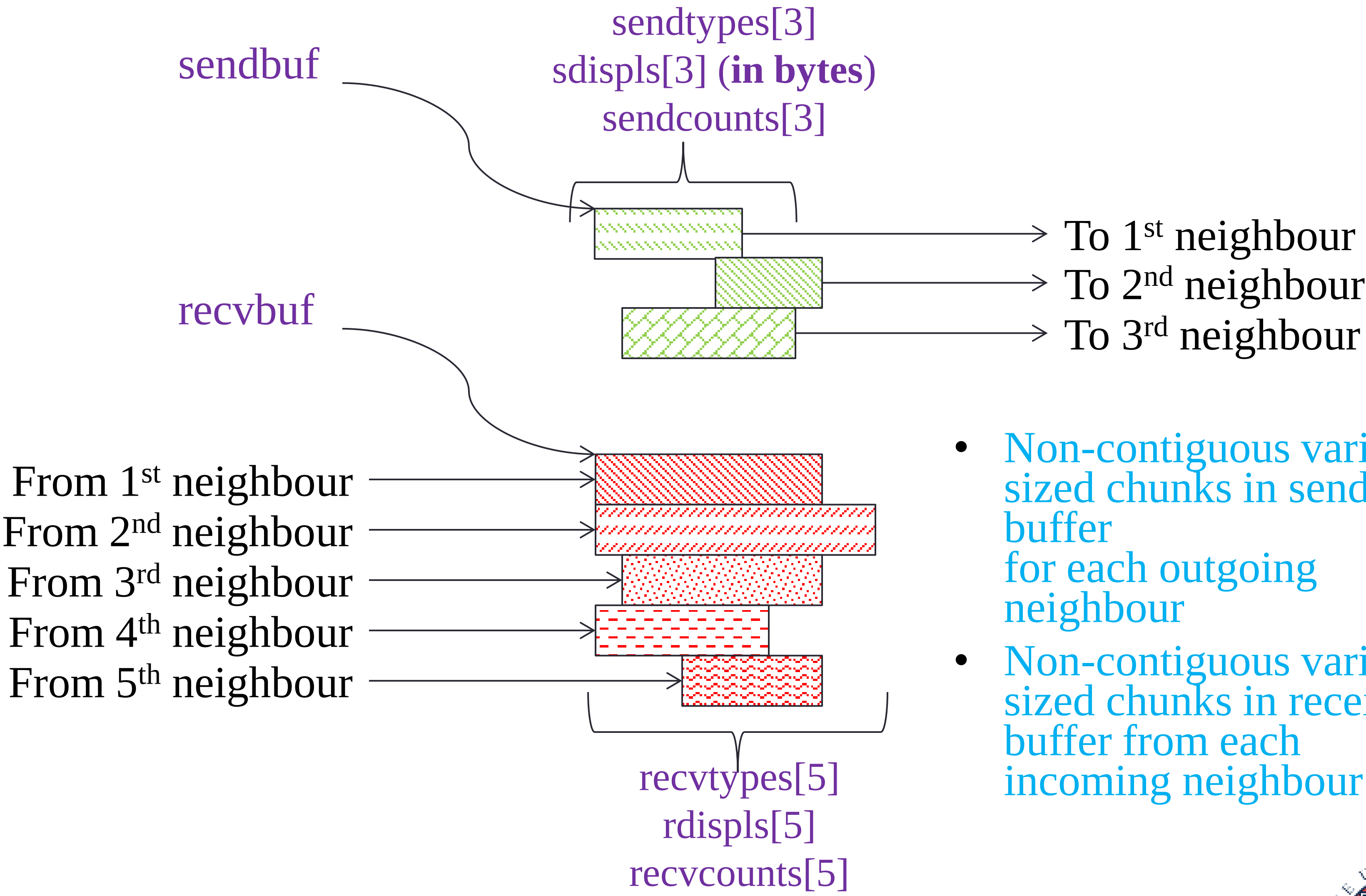
MPI_Neighbor_alltoallv



- Non-contiguous variable-sized chunks in send buffer
  for each outgoing neighbour
- Non-contiguous variable-sized chunks in receive buffer from each incoming neighbour

epcc

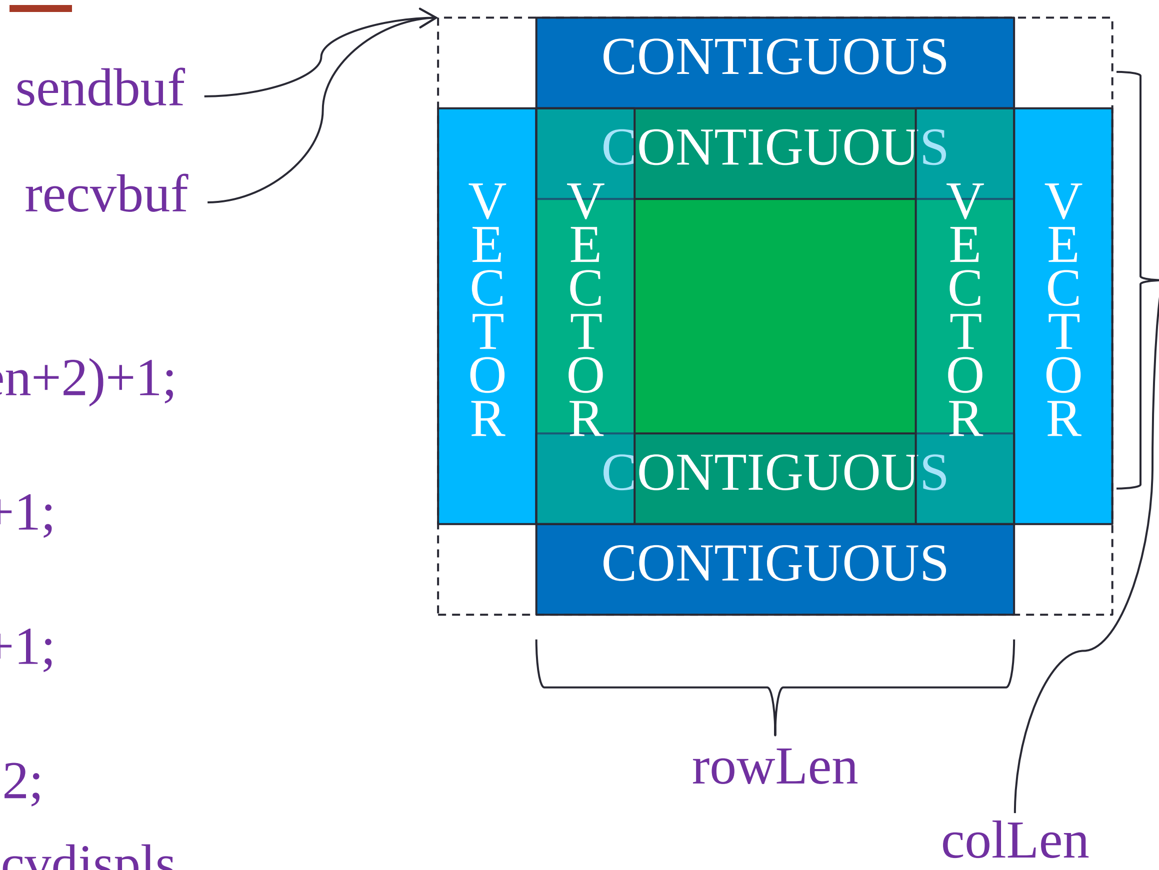
MPI_Neighbor_alltoallw



- Non-contiguous variable-sized chunks in send buffer
  for each outgoing neighbour
- Non-contiguous variable-sized chunks in receive buffer from each incoming neighbour

MPI_Neighbor_alltoallw

```
for (int i=0;i<4;++i) {
    sendcounts[i] = 1;
    recvcounts[i]=1; }

sendtypes[0] = contigType;
senddispls[0] = colLen*(rowLen+2)+1;
sendtypes[1] = contigType;
senddispls[1] = 1*(rowLen+2)+1;
sendtypes[2] = vectorType;
senddispls[2] = 1*(rowLen+2)+1;
sendtypes[3] = vectorType;
senddispls[3] = 2*(rowLen+2)-2;

// similarly for recvtypes and recvdispls

MPI_Neighbor_alltoallw(sendbuf, sendcounts, senddispls, sendtypes,
                       recvbuf, recvcounts, recvdsipls, recvtypes,
                       comm);
```



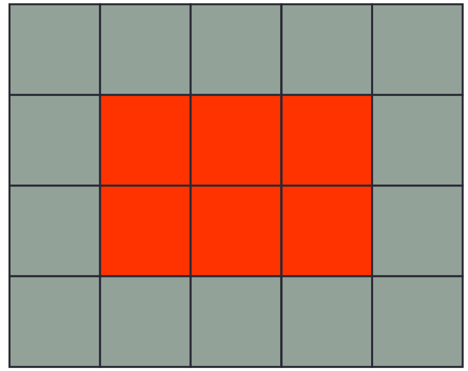
Why bytes for Alltoallw displs?

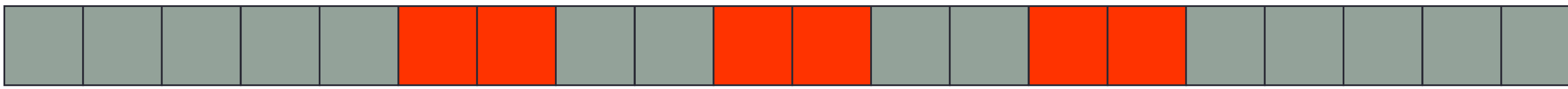
- Normally, displacements are in number of objects
  - MPI hates talking about bytes!

- Byte offset = displ * extent(object)
  - but what is the extent of a datatype with holes?
  - and is it useful?

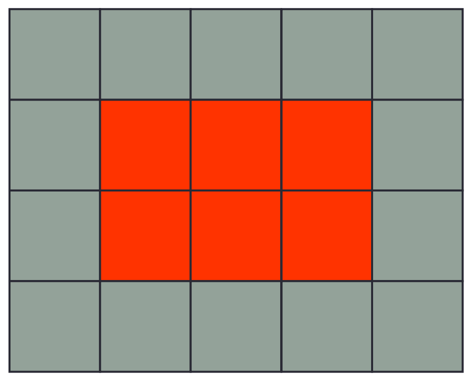
Array Subsections in Memory

C: **`x[5][4]`**

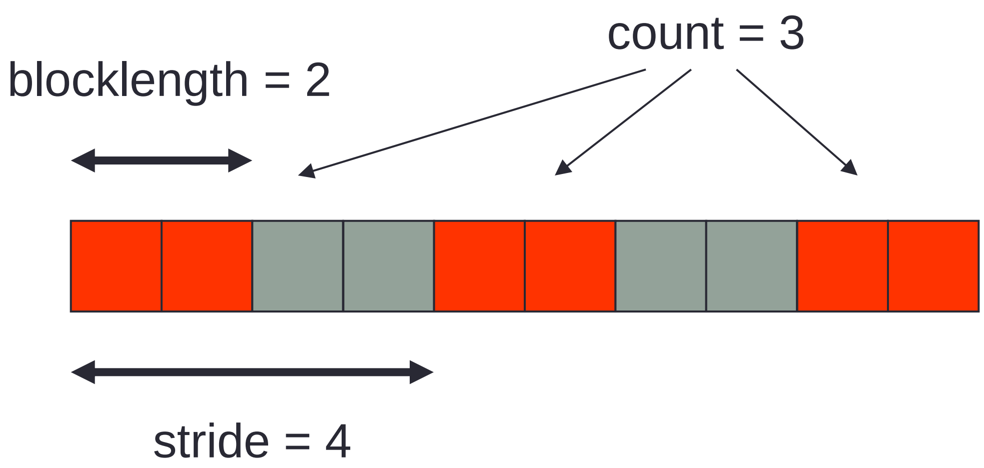


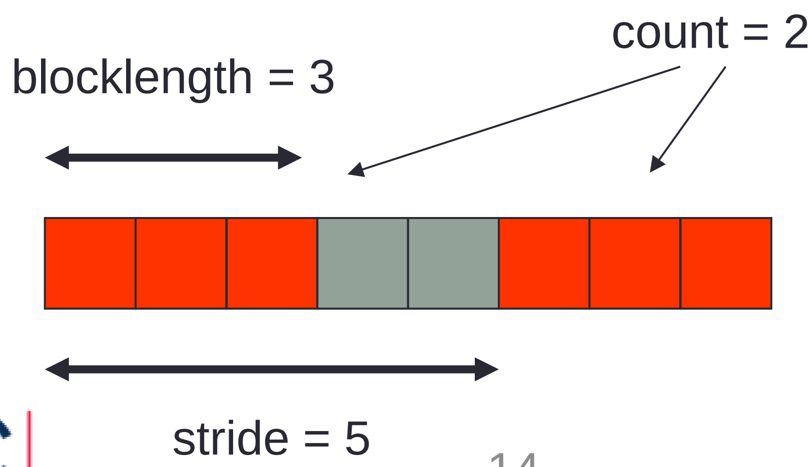


F: **`x(5,4)`**



Equivalent Vector Datatypes





Definition in MPI

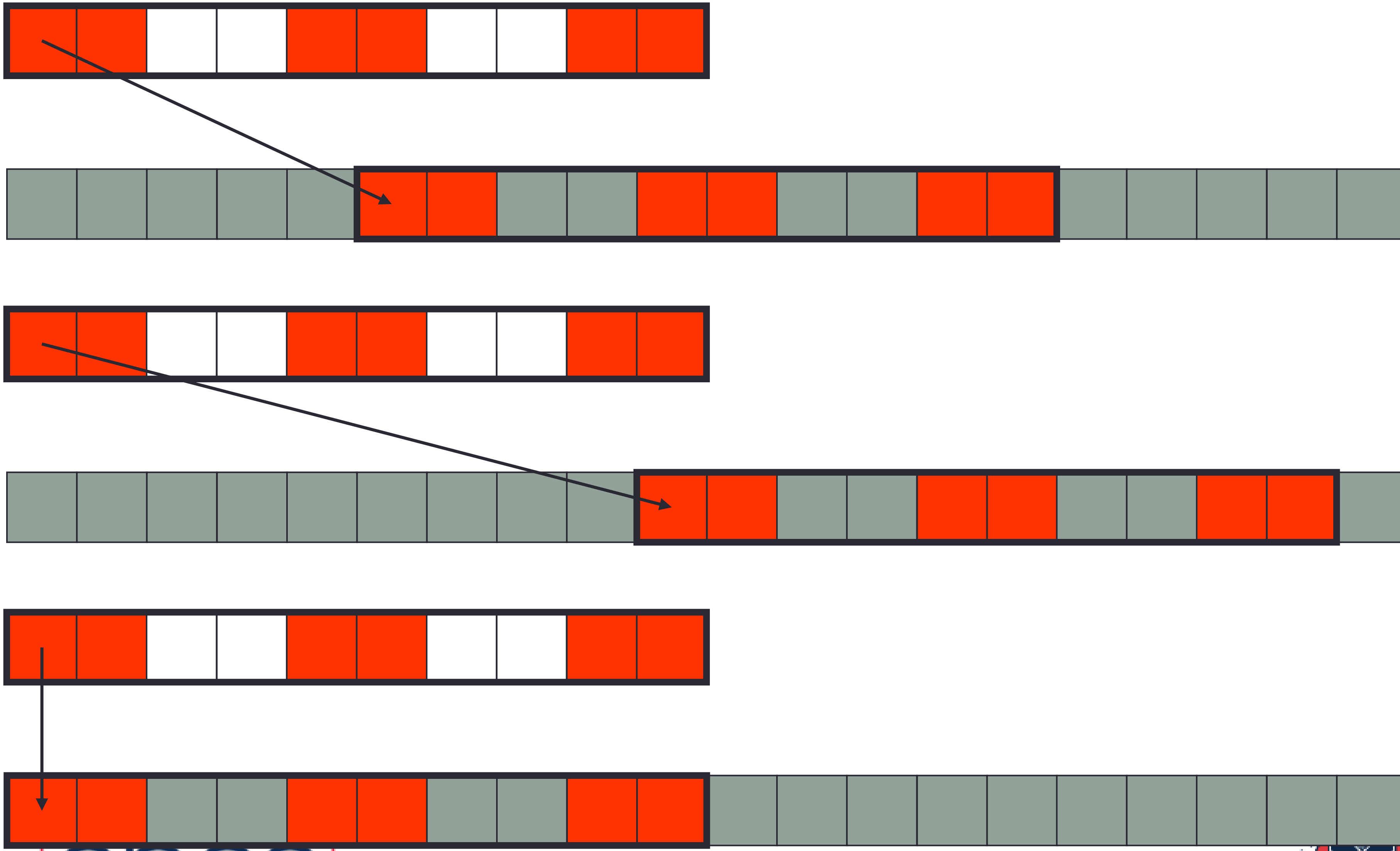
```
MPI_Type_vector(int count, int blocklength, int stride,
            MPI_Datatype oldtype, MPI_Datatype *newtype);
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,
                  OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE
INTEGER NEWTYPE, IERR
```

```
MPI_Datatype vector3x2;
MPI_Type_vector(3, 2, 4, MPI_FLOAT, &vector3x2)
MPI_Type_commit(&vector3x2)
```

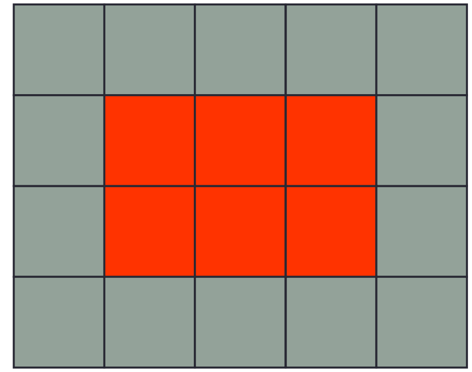
```
integer vector3x2
call MPI_TYPE_VECTOR(2, 3, 5, MPI_REAL, vector3x2, ierr)
call MPI_TYPE_COMMIT(vector3x2, ierr)
```

Datatypes as Floating Templates

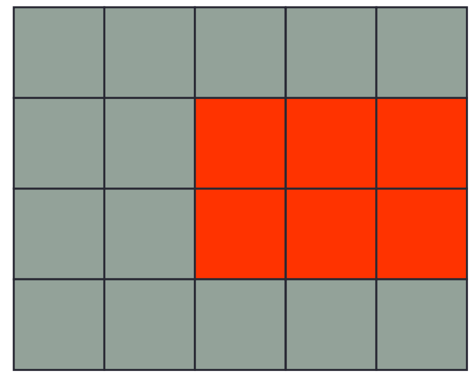


Choosing the Subarray Location

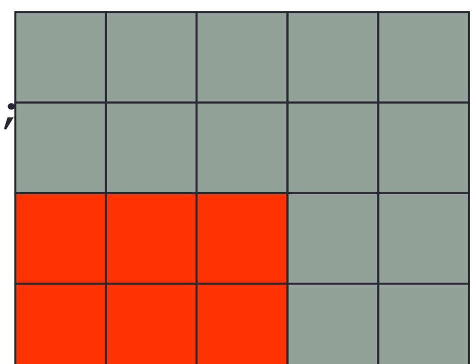
```
MPI_Send(&x[1][1], 1, vector3x2, ...);
MPI_SEND(x(2,2)  , 1, vector3x2, ...)
```



```
MPI_Send(&x[2][1], 1, vector3x2, ...);
MPI_SEND(x(3,2)  , 1, vector3x2, ...)
```

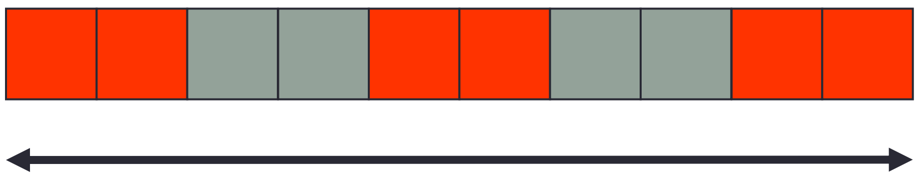


```
MPI_Send(&x[0][0], 1, vector3x2, ...);
MPI_SEND(x(1,1)  , 1, vector3x2, ...)
```



Datatype Extents

- When sending multiple datatypes
  - datatypes are read from memory separated by their extent
  - for basic datatypes, extent is the size of the object
  - for vector datatypes, extent is distance from first to last data



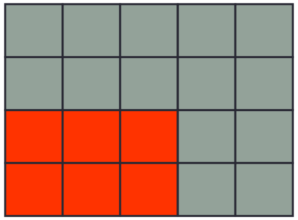
extent = 10*extent(basic type)

extent = 8*extent(basic type)

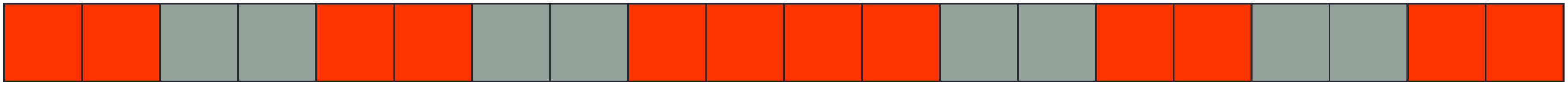
- Extent *does not* include trailing spaces

Sending Multiple Vectors

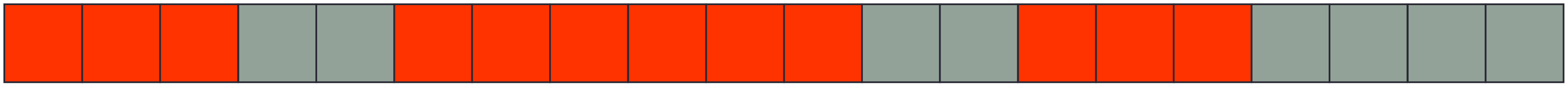
```
MPI_Send(&x[0][0], 1, vector3x2, ...);

MPI_SEND(x(1,1)   , 1, vector3x2, ...)
```



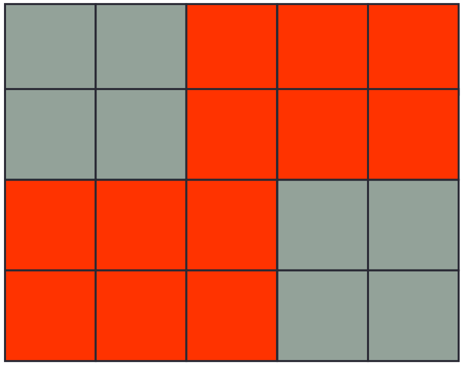
```
MPI_Send(&x[0][0], 2, vector3x2, ...);
```



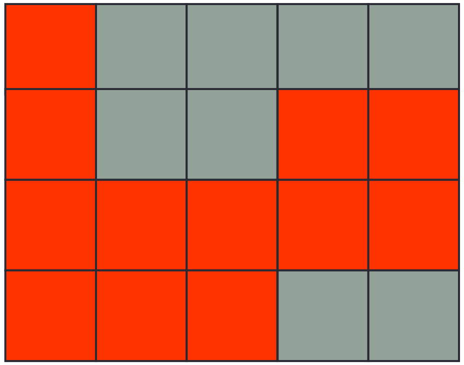
```
MPI_SEND(x(1,1)   , 2, vector3x2, ...)
```



C



F



Issues with Vectors

- Sending multiple vectors is not often useful
  - extents are not defined as you might expect for 2D arrays
- A 3D array subsection is not a vector
  - but cannot easily use 2D vectors as building blocks due to extents
  - becomes even harder for higher-dimensional arrays
- It is possible to set the extent manually
  - routine is called **`MPI_Type_create_resized`**
- For example, difficult to use vectors with **`MPI_Scatter`** to scatter 2D datasets

MPI_Scatter 2D array

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

- Problem (i): displacements are not constant
  - here, offsets from origin are 0, 2, 8 and 10 (floats)
- Solution
  - use **`MPI_Scatterv`** which takes separate displacement for each rank
- Problem (ii): displacements multiplied by extent = 6 floats
  - required offsets are not an integer multiple of the extent!
- Solution
  - use **`MPI_Type_create_resized`** to reset extent to, e.g., one float

So why bytes for Alltoallw displs?

- Alltoall
  - one datatype and no displacements
  - byte displacement of message “i” is extent(dataype)*i

- Alltoallv
  - one datatype and multiple displacements
  - byte displacement of message “i” is extent(dataype[i])*i
  - enables halo swapping in CFD exercise
  - but a 2D decomposition has contiguous and non-contiguous halos

- Alltoallw
  - multiple datatypes and multiple displacements
  - I give up – work out the byte displacements yourself!

epcc

Summary

- Regular or irregular domain decomposition codes
  - Where the decomposition is static or changes infrequently
- Should investigate replacing point-to-point communication
  - E.g. halo-exchange communication
- With neighbourhood collective communication
  - Probably MPI_Ineighbor_alltoallw
- So that MPI can optimise the whole pattern of messages
  - Rather than trying to optimise each message individually
- And so your application code is simpler and easier to read