MPI 3.0 Neighbourhood Collectives

Advanced Message-Passing Programming

Engineering and Physical Sciences Research Council

Natural Environment Research Council

Overview

- Review of topologies in MPI
- MPI 3.0 added 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:
  - Replace all point-to-point halo-exchange communication with a single neighbourhood collective in your MPP coursework code

Topologies

- Imagine 2D domain decomposition of an *L* x *L* array
  - domain split up into *P* subdomains of size *L*/*Px* x *L*/*Py* , *Px* * *Py* = *P*
  - nearest-neighbour interaction implies nearest-neighbour comms
  - results in a 2D grid of *Px* x *Py* processes (which swap halos)
- Decomposition of unstructured mesh of *N* elements
  - domain split up into *P* subdomains each of *N*/*P* elements
  - nearest-neighbour interaction implies nearest-neighbour comms
  - results in a general graph of *P* processes (which swap halos)
    - each process communicates with an arbitrary number of neighbours
  - can be weighted: vertex = computation cost, edges = comms load
  - comms graphs typically undirected
    - if *A* communicates with *B* then *B* communicates with *A*

Topology communicators

- 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 both their incoming and outgoing neighbours
      - incoming and outgoing the same for undirected graph
  - MPI_DIST_GRAPH_CREATE
    - any process can specify any edge in the graph (too general?)
    - only need to specify outgoing neighbours
      - MPI library must do communication to work out the global pattern

Topology communicators

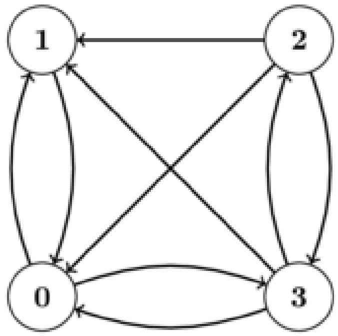
- Testing the topology type associated with a communicator
  - MPI_TOPO_TEST
- Finding my neighbours in a cartesian topology
  - MPI_CART_SHIFT

  - Find out how many neighbours there are of any process
    - MPI_GRAPH_NEIGHBORS_COUNT
  - Get the ranks of all neighbours of any process
    - MPI_GRAPH_NEIGHBORS
  - Find out how many neighbours I have
    - MPI_DIST_GRAPH_NEIGHBORS_COUNT
  - Get the ranks of all my neighbours
    - MPI_DIST_GRAPH_NEIGHBORS

Example

- Useful example program at: https://riptutorial.com/mpi/example/29195/graph-topology-creation-and-communication

Creates a graph topology in a distributed manner so that each node defines its neighbors. Each node communicates its rank among neighbors with `MPI_Neighbor_allgather`.



Example (cont)

```
#include <mpi.h>
#include <stdio.h>

#define nnode 4

int main()
{
    MPI_Init(NULL, NULL);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int source = rank;
    int degree;
    int dest[nnode];
    int weight[nnode] = {1, 1, 1, 1};
    int recv[nnode] = {-1, -1, -1, -1};
    int send = rank;

    // set dest and degree.
    if (rank == 0)
    {
        dest[0] = 1;
        dest[1] = 3;
        degree = 2;
    }
    else if(rank == 1)
    {
        dest[0] = 0;
        degree = 1;
    }
    else if(rank == 2)
    {
        dest[0] = 3;
        dest[1] = 0;
        dest[2] = 1;
        degree = 3;
    }
    else if(rank == 3)
    {
        dest[0] = 0;
        dest[1] = 2;
        dest[2] = 1;
        degree = 3;
    }

    // create graph.
    MPI_Comm graph;
    MPI_Dist_graph_create(MPI_COMM_WORLD, 1, &source, &degree, dest, weight,
                          MPI_INFO_NULL, 1, &graph);

    // send and gather rank to/from neighbors.
    MPI_Neighbor_allgather(&send, 1, MPI_INT, recv, 1, MPI_INT, graph);

    printf("Rank: %i, recv[0] = %i, recv[1] = %i, recv[2] = %i, recv[3] = %i\n",
        rank, recv[0], recv[1], recv[2], recv[3]);

    MPI_Finalize();
    return 0;
}

// Taken from https://riptutorial.com/mpi/example/29195/graph-topology-creation-and-communication
```

Reordering

- Reorder = true enables remapping of processes
  - e.g. try to place neighbours on the same node
    - minimise number of inter-node communications over the network

- Can also take into account the weights
  - equal computational load on each node
  - minimise communications volume across network

- Interesting to see if / how well this is done in practice ...

Process Distribution (i)

- Imagine running 256 MPI processes on 4 nodes
  - each node has 64 CPU-cores
  - almost all systems put ranks 0-63 on node 0, 63-127 on node 1, ...
- But this may not be optimal!

Process Distribution (ii)

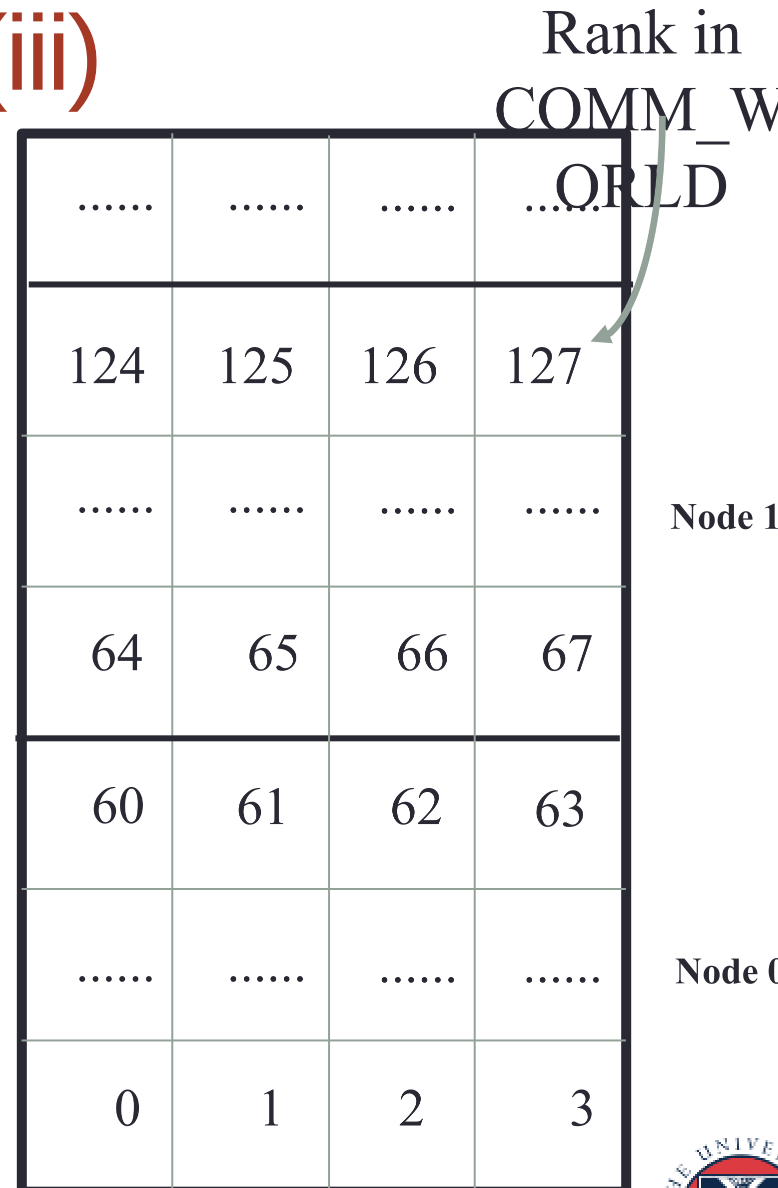
- We have a 64 x 1024 array
  - create a cyclic 2D Cartesian Communicator on 256 MPI processes
  - choose a 4 x 64 distribution so each local domain is 16 x 16 square
- Each process communicates with its 4 nearest neighbours
  - 128 messages sent over the network from each node

63	127	191	255
62	126	190	254
61	125	189	253
……	……	……	……
2	66	130	194
1	65	129	193
0	64	128	192
Node 0	**Node 1**	**Node 2**	**Node 3**

epcc

Process Distribution (iii)

- Switching the process axes is much better
  - 8 messages per node over network
- But how do we achieve this after our program has started?
- Set reorder = TRUE
  - hope rank 60 in COMM_WORLD becomes rank 2 in COMM_CART
    - and 3 becomes 192
    - ...
- Or do the remapping by hand
  - using MPI_Comm_split()



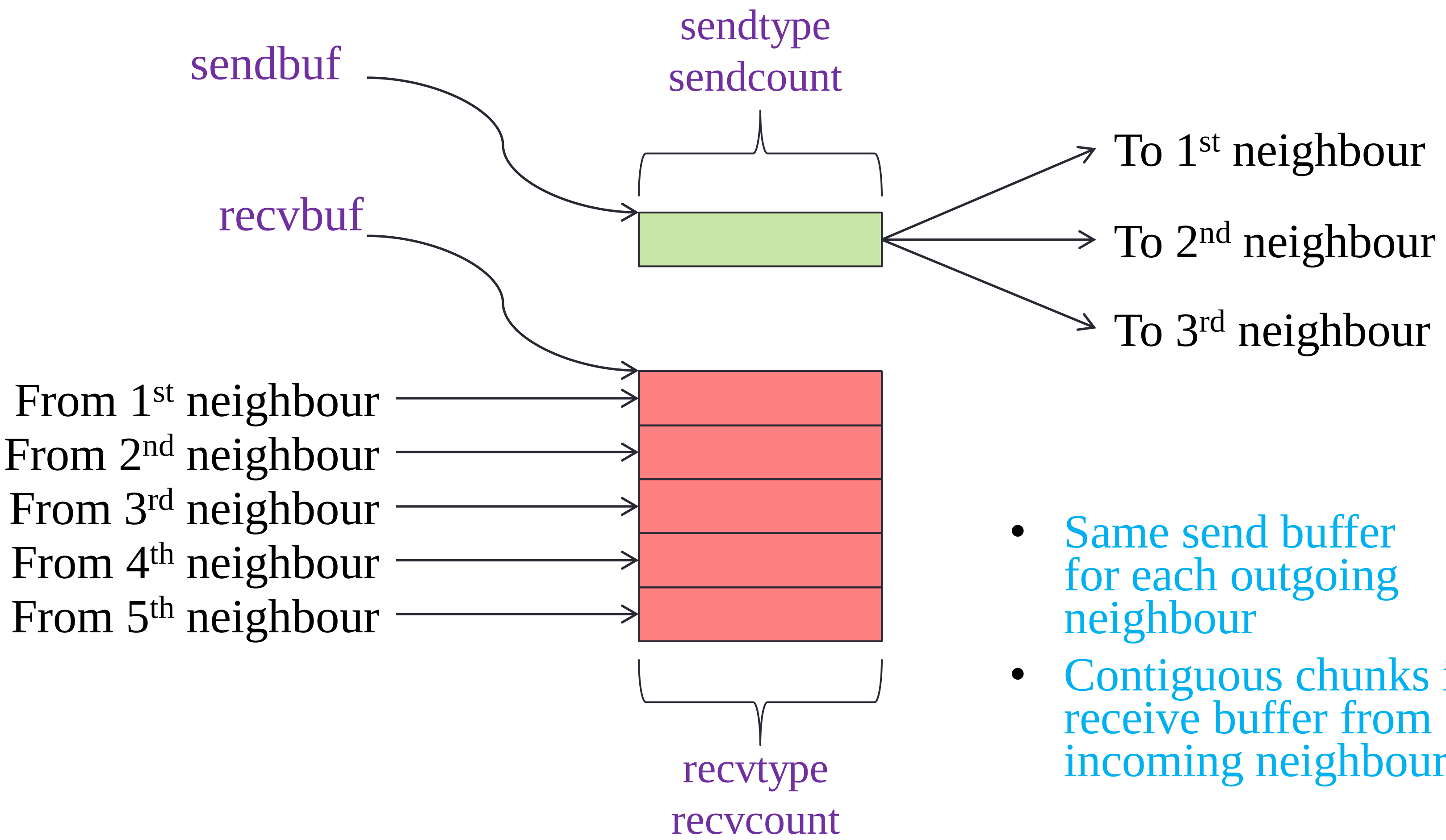
Job launcher options

- Reordering is just a logical change of rank
  - actual MPI process doesn't move
  - might require you to exchange data between new and old ranks
- Sometime easier to do remapping at launch time
  - change default allocation of processes -> CPU-cores rather than accepting default and remapping within the MPI program
- SLURM
  - srun has many (complicated) options for this - see manual for details!
- Tools can help here
  - e.g. HPE "perftools" on ARCHER2 can analyse inter-process communications and suggest an optimal mapping

Neighbourhood collective operations

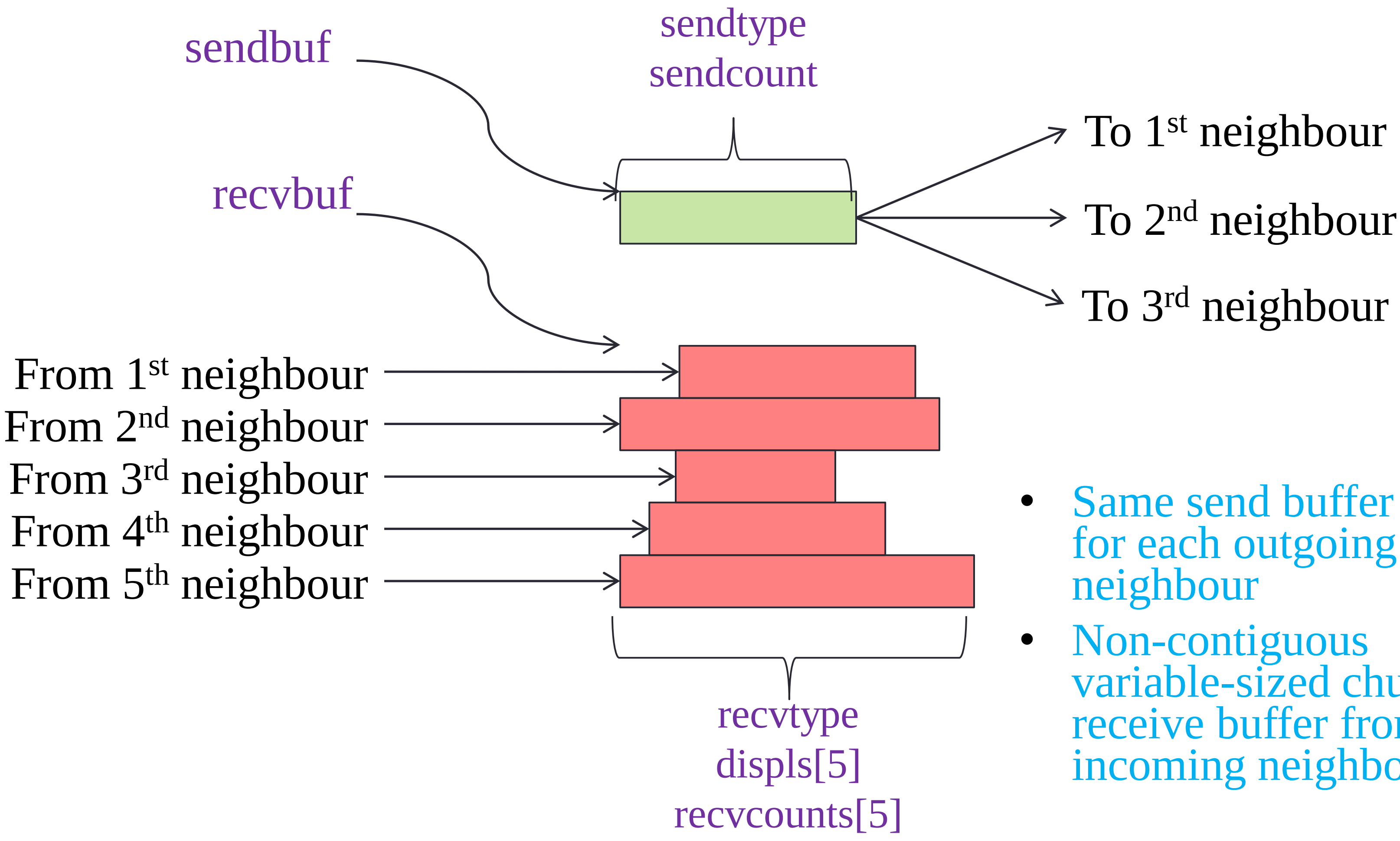
- See section 8.6 in MPI 4.0 for blocking functions
  - See section 8.7 in MPI 4.0 for non-blocking functions
  - See section 8.8 in MPI 4.0 for an example application
- MPI_[N|In]eighbor_allgather[v]
  - Send same piece of data to all neighbours
  - Gather one piece of data from each neighbour
- MPI_[N|In]eighbor_alltoall[v|w]
  - Send different data to each neighbour
  - Receive different data from each neighbour
- Use-case: regular or irregular domain decompositions
  - Where the decomposition is static or changes infrequently
  - Because creating a topology communicator takes time

MPI_Neighbor_allgather



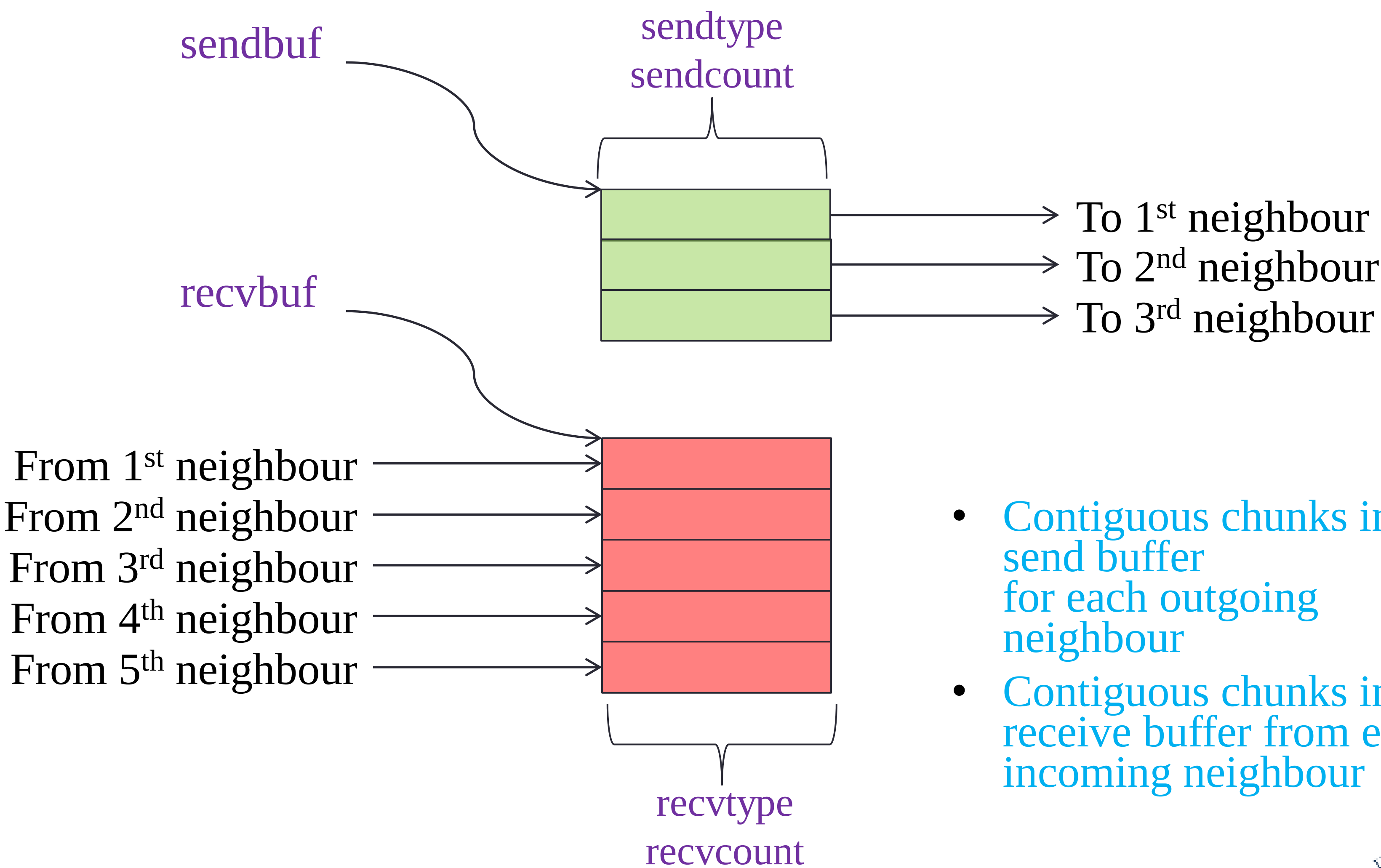
- Same send buffer for each outgoing neighbour
- Contiguous chunks in receive buffer from each incoming neighbour

MPI_Neighbor_allgatherv



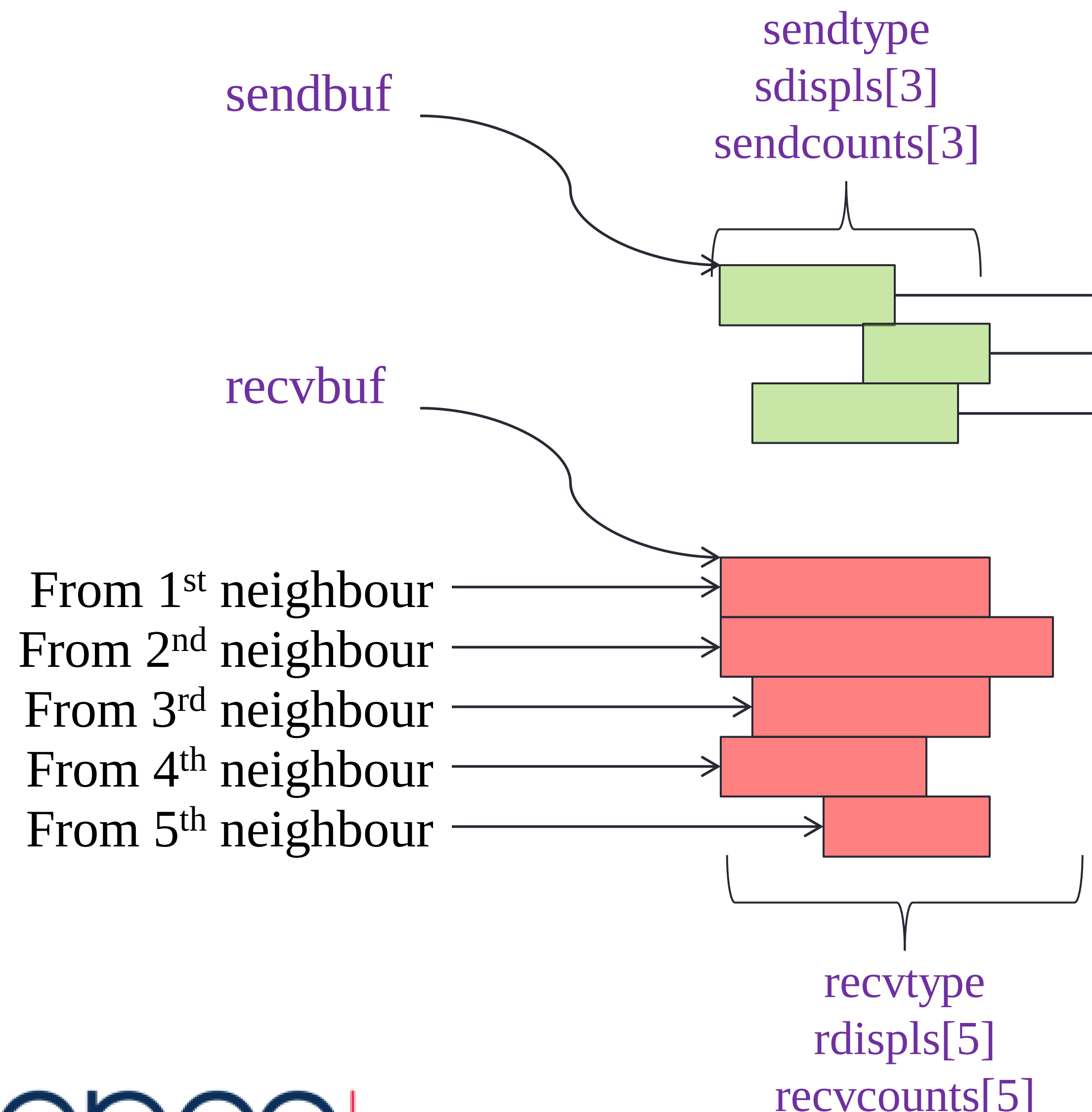
- Same send buffer for each outgoing neighbour
- Non-contiguous variable-sized chunks in receive buffer from each incoming neighbour

MPI_Neighbor_alltoall



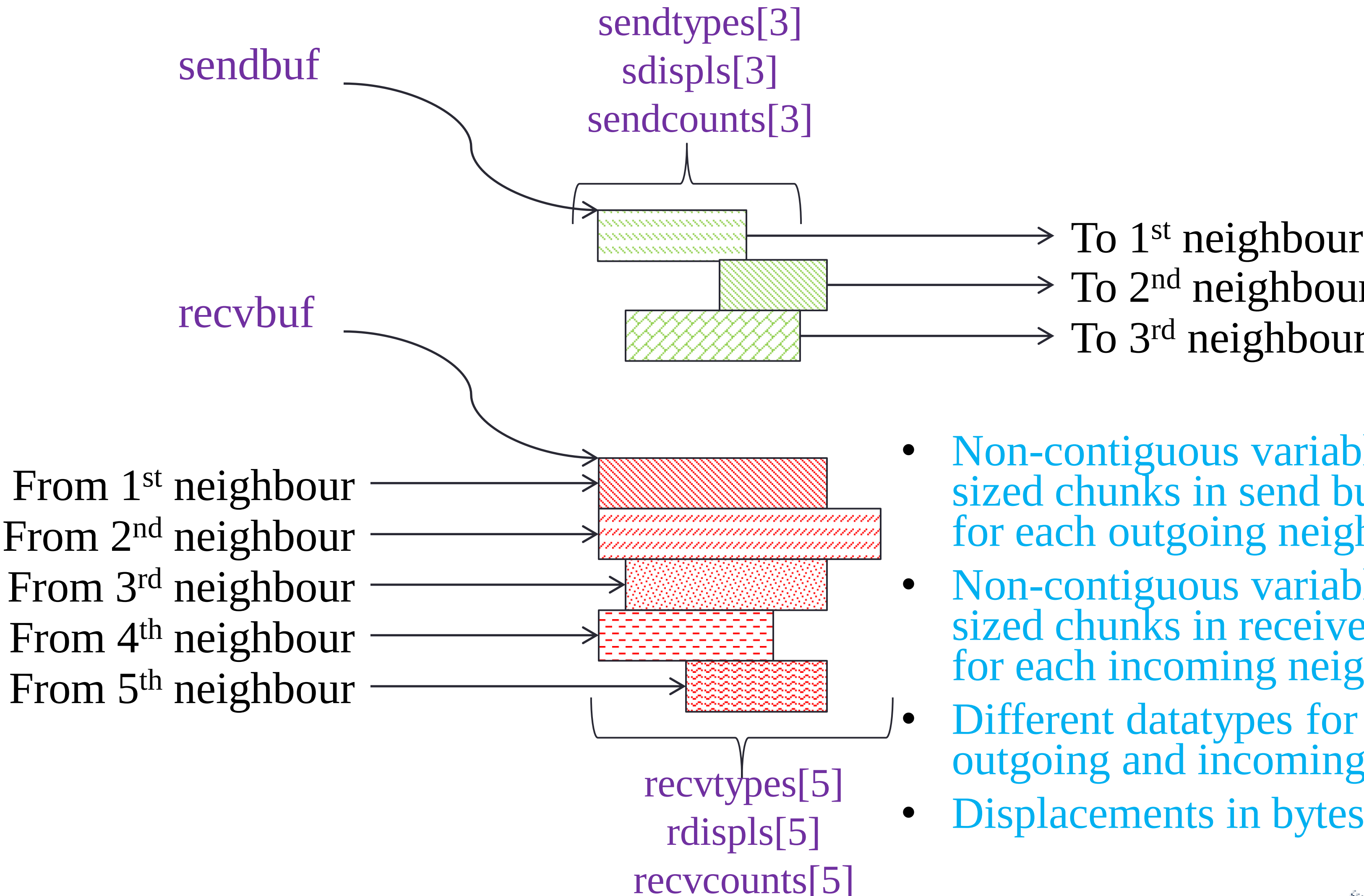
- Contiguous chunks in send buffer for each outgoing neighbour
- Contiguous chunks in receive buffer from each incoming neighbour

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

MPI_Neighbor_alltoallw



- Non-contiguous variable-sized chunks in send buffer for each outgoing neighbour
- Non-contiguous variable-sized chunks in receive buffer for each incoming neighbour
- Different datatypes for every outgoing and incoming chunk
- Displacements in bytes

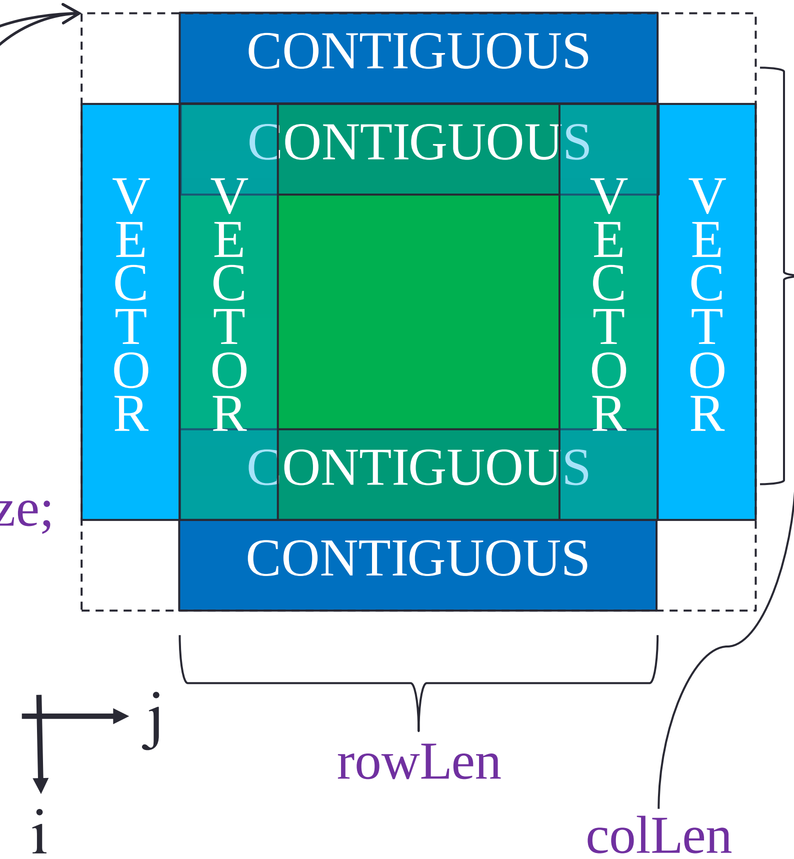
MPI_Neighbor_alltoallw

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

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

// similarly for recvtypes and recvdispls

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



Summary

- Useful for regular or irregular domain decomposition
  - Where the decomposition is static or changes infrequently
- Investigate replacing point-to-point communication
  - E.g. halo-exchange communication
- With neighbourhood collective communication
  - Probably MPI_Neighbor_alltoallw / 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