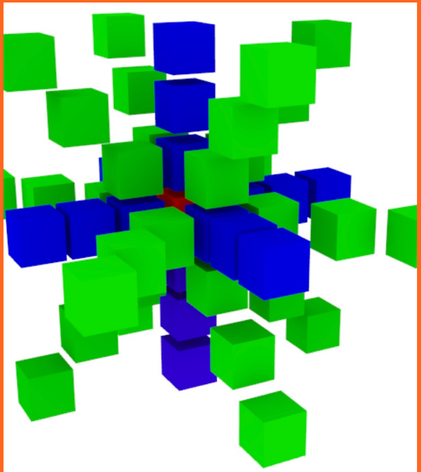
CS-E4690 – Programming Parallel Supercomputers MPI: Collective communications and advanced topics

Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi

Aalto University
School of Science

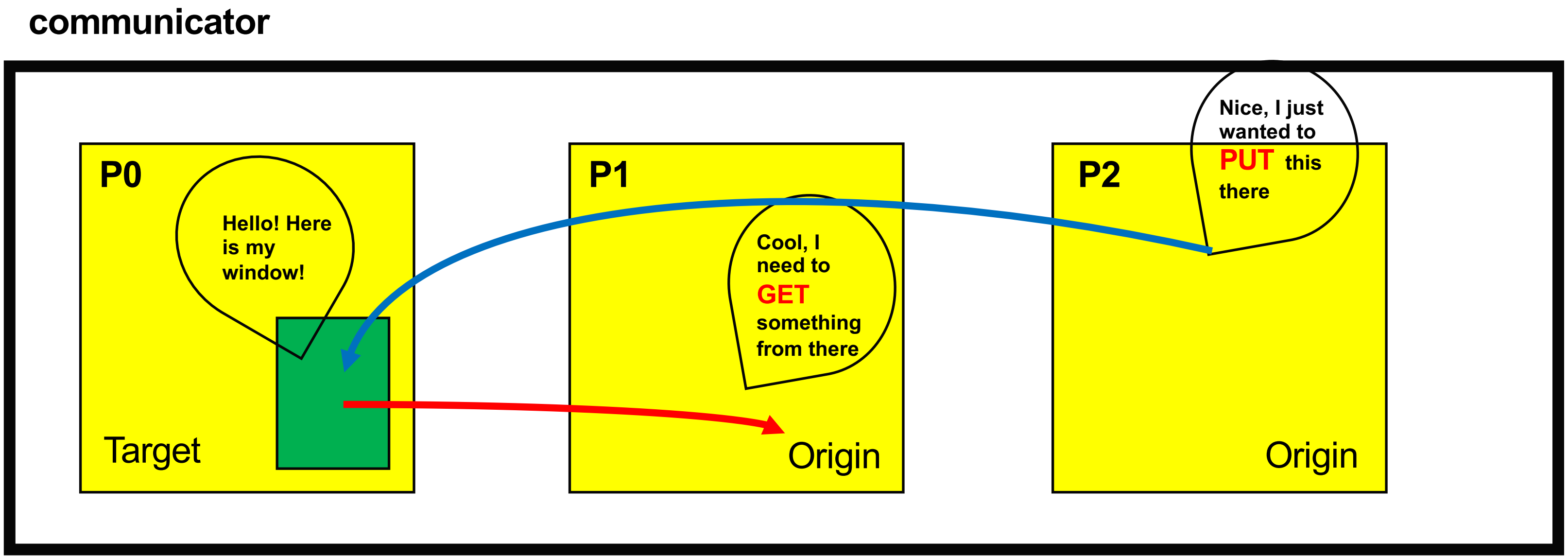


One-sided communication

Remote memory reads and writes (requires RMA technology, but nowadays standardly available on multiprocessors chips)

- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- Can, in principle, reduce synchronization overheads

Window



Origins could also wish to ACCUMULATE data to/from target process, which can include a reduction operation.

Typical workflow

```
MPI_Info info;
MPI_Win window;
MPI_Win_create( /* size info */, info, comm, &memory, &window );
// do put and get calls
MPI_Win_free( &window );
```

Window properties

- Create window call is collective (all in certain comm must call it)
- The window size can be set individually on each process (also to null)
- The window location can be set individually on each process
- The window is the **target** of data in a **put** operation, or the **source** of data in a **get** operation.
- There can be memory associated with a window, so it needs to be freed explicitly with MPI_Win_free(win).

Creating a window (basic)

MPI_Alloc_mem to allocate the “base” buffer

int MPI_Win_create (void *base, MPI_Aint size, int disp_unit,
MPI_Info info, MPI_Comm comm, MPI_Win *win)

base (pointer to) local memory to expose for RMA

size of a local window in bytes
disp_unit local unit size for displacements in bytes

info info argument

comm communicator
win handle to window

MPI_Win_free(win)

Creating a window (with allocate)

int MPI_Win_allocate (MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)

Allocates window segment

size: size of a local window in bytes

disp_unit local unit size for displacements, in bytes

info: info argument

comm: communicator

baseptr: address of local allocated window segment

win: window object returned by the call

MPI_Win_free(win)

Creating a window (dynamic)

int MPI_Win_create_dynamic (MPI_Info info, MPI_Comm comm, MPI_Win *win)

Similar to the others, but only a pointer to a window object, with its attached memory yet unspecified, is returned.

to an empty buffer is returned. To attach memory to the Window:

MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
MPI_Win_detach(MPI_Win win, void *base)

Advanced MPI, finding out details left for interested students

MPI_Win_free(win)

Synchronization?

Active synchronization:

- Both origin and target process perform synchronization calls

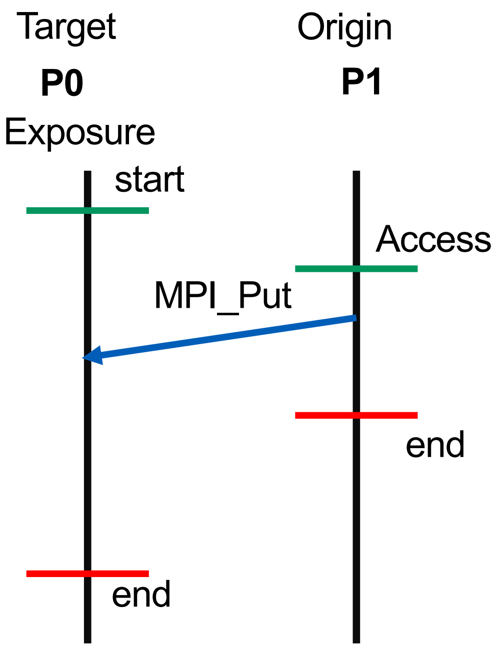
Passive synchronization:

- No synchronization calls at the target

Active mode:

Communication takes place within **epochs**

- **Exposure epoch** on target, **access epoch** on origin
- Synchronization calls start and end an epoch
- An epoch is specific to a particular window



Epoch: time in between the green and red lines

Active (global) RMA synchronization: fences

MPI_Win_fence(assert, win)

assert optimize for specific usage. Valid values are ”0”,

MPI_MODE_NOSTORE, MPI_MODE_NOPUT,
MPI_MODE_NOPRECEDE, MPI_MODE_NOSUCCEED

win window handle

- **Used both starting and ending an epoch**
- **Assertions with 0 will always work, but being more specific could help, see the next slide (advanced)**

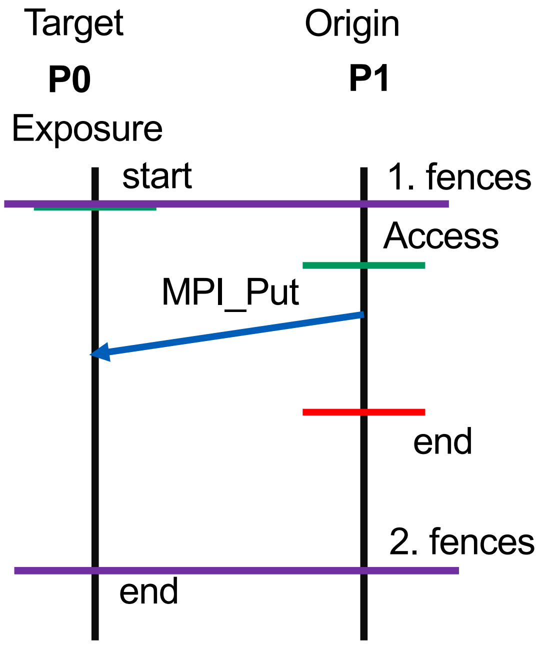
MPI/One_sided_1.c

Assertions for fence ops (advanced)

- MPI_MODE_NOSTORE the local window was not updated by local stores (or local get or receive calls) since last synchronization.
- MPI_MODE_NOPUT the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
- MPI_MODE_NOPRECEDE the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
- MPI_MODE_NOSUCCEED the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.
- Specifying these may help in optimizing the communication.

Fenced synchronization is restrictive

- GLOBAL: every process has to execute the calls
- Fences act as Barrier-like operations; recall the BSP model
- Can be inefficient
- The need for global synchronization can be avoided by defining processor groups; this requires additional knowledge about the communicators, and we will come back to this later on



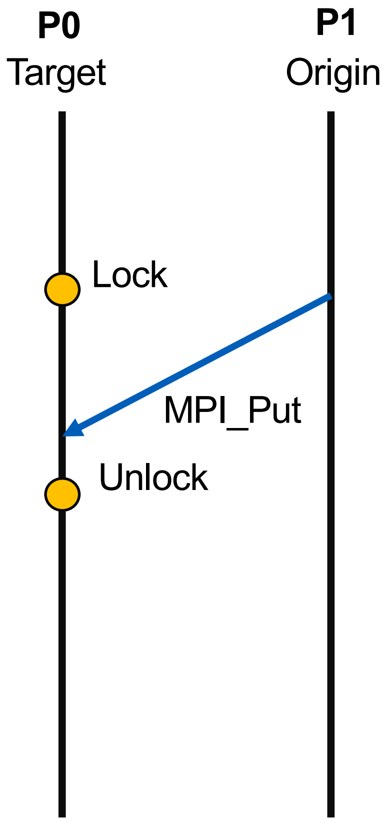
Epoch: time in between the purple lines

Passive synchronization

MPI/One_sided_3.c

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

lock_type: Indicates whether other processes may access the target window at the same time (if MPI_LOCK_SHARED) or not (MPI_LOCK_EXCLUSIVE)
rank: rank of the process having the **locked** (target) window
assert: Used to optimize this call; zero may be used as a default.
win: window object



int MPI_Win_unlock(int rank, MPI_Win win)

Moving data: put

```
int MPI_Put( const void *origin_addr, int origin_count,
        MPI_Datatype origin_datatype, int target_rank, MPI_Aint
        target_disp, int target_count, MPI_Datatype target_datatype,
        MPI_Win win)
```

- Otherwise very normal-looking call, but the target data description is somewhat non-trivial
- When creating a window, you need to specify the displacement unit (from the window start)

MPI/One_sided_1.c

Moving data: get

int MPI_Get(const void *origin_addr, int origin_count,

MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

- Similar syntax to MPI_Put

Moving data: accumulate

int MPI_Accumulate (const void *origin_addr, int origin_count,MPI_Datatype origin_datatype, int target_rank,MPI_Aint target_disp, int target_count,MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

- Store data from the **origin** process to the memory window of the **target** process *and* combine it using one of the predefined MPI reduction operations
- **Predefined** operators are available (we talk about these in the connection of collectives), but **no user-defined ones**.
- There is one extra operator: **MPI_REPLACE**, this has the effect that only the last result to arrive is retained.

Moving data: get_accumulate

int MPI_Get_accumulate (const void *origin_addr, int origin_count,MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

- Store data from target window to the origin, and combine it with the predefined operation.
- **Predefined** operators are available (we talk about these in the connection of collectives), but **no user-defined ones**.
- There is one extra operator: **MPI_REPLACE**, this has the effect that only the last result to arrive is retained.

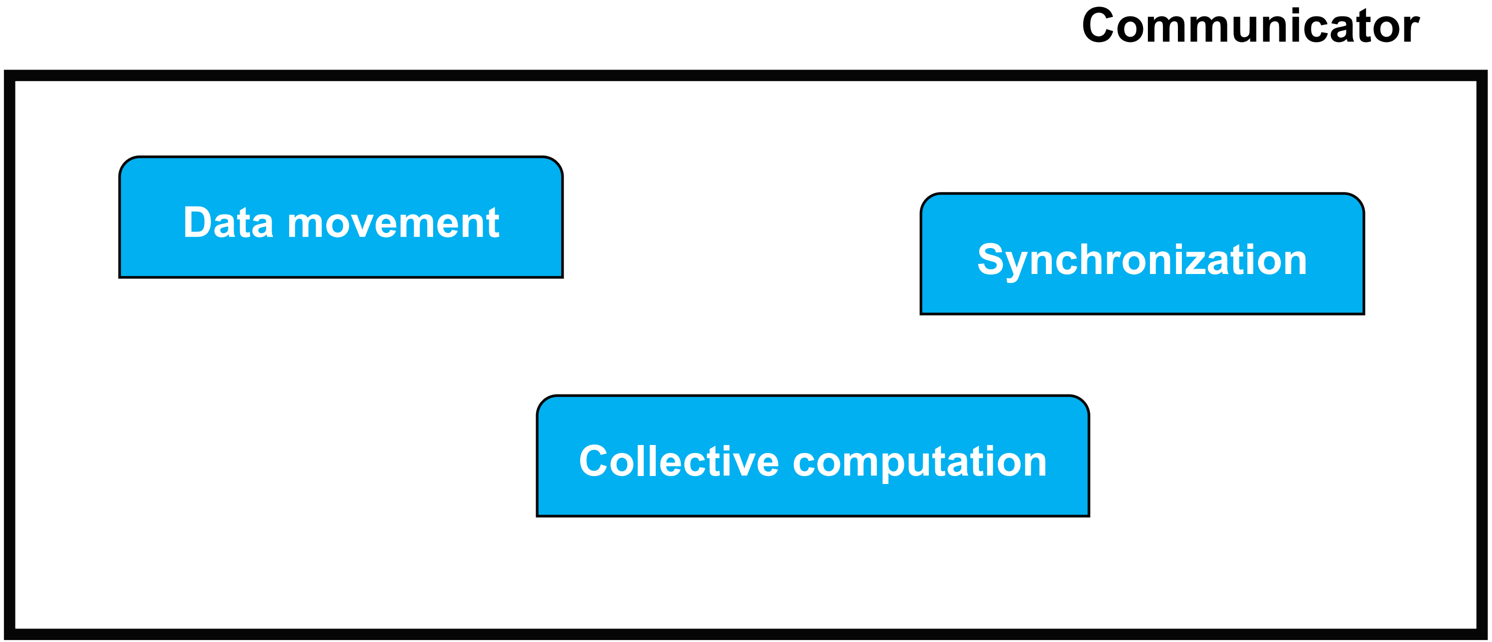
Note: many processes!

- **Within an epoch**, no guaranteed ordering of Get and Put operations: if you make many such calls in a mixture, there is no guarantee who gets to overlapping data first, and the results may turn out to be garbage (*race conditions*).
- Accumulates are ‘atomic’:
  - MPI_Accumulate with MPI_REPLACE implements an atomic put that has a well-defined order
  - MPI_Get_accumulate with MPI_NO_OP implements an atomic get that has a well-defined order.
- **Multiple atomic operations are safe within an epoch.**

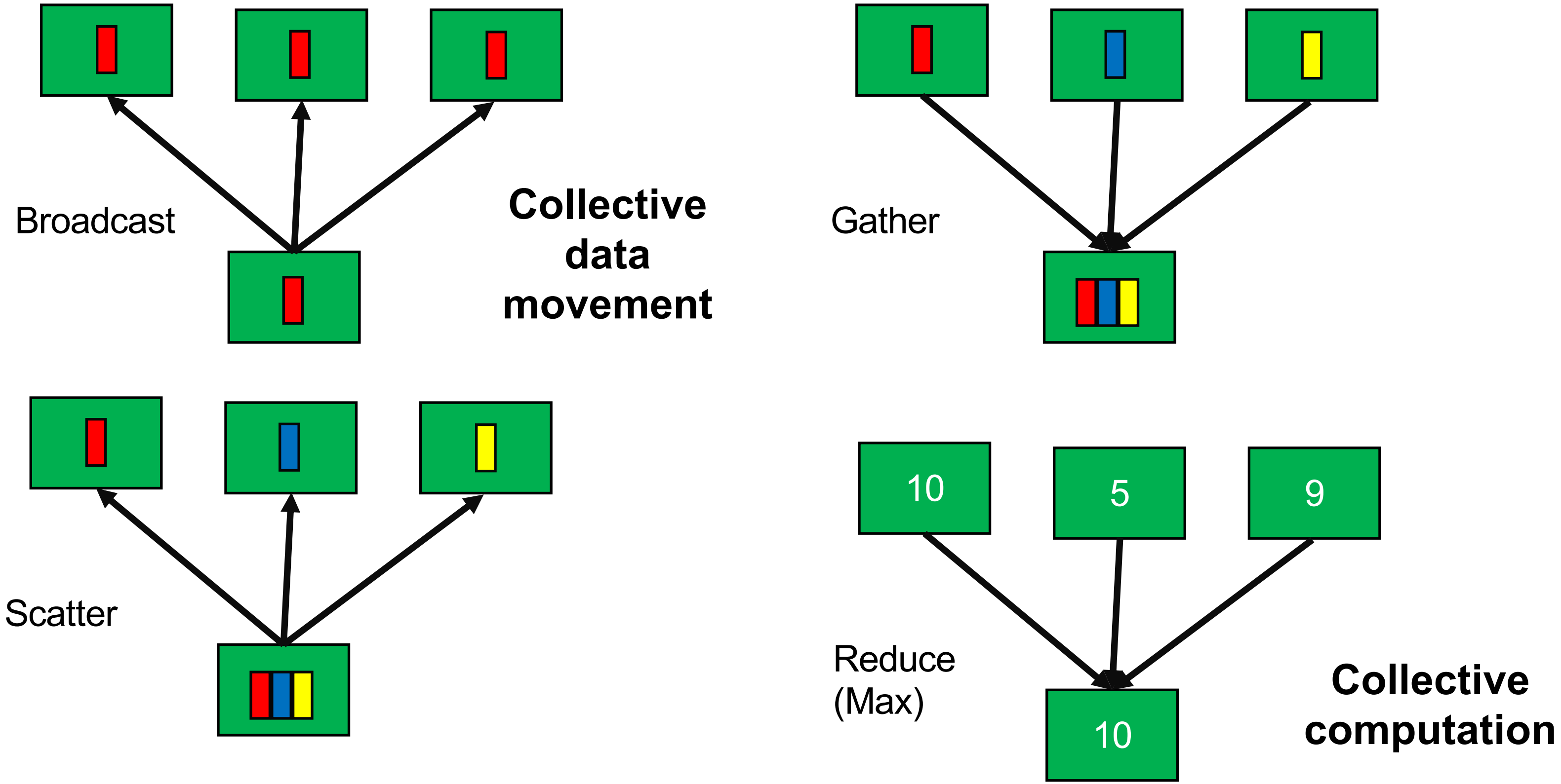
MPI/One_sided_1.c

MPI/One_sided_2.c

Collective communications



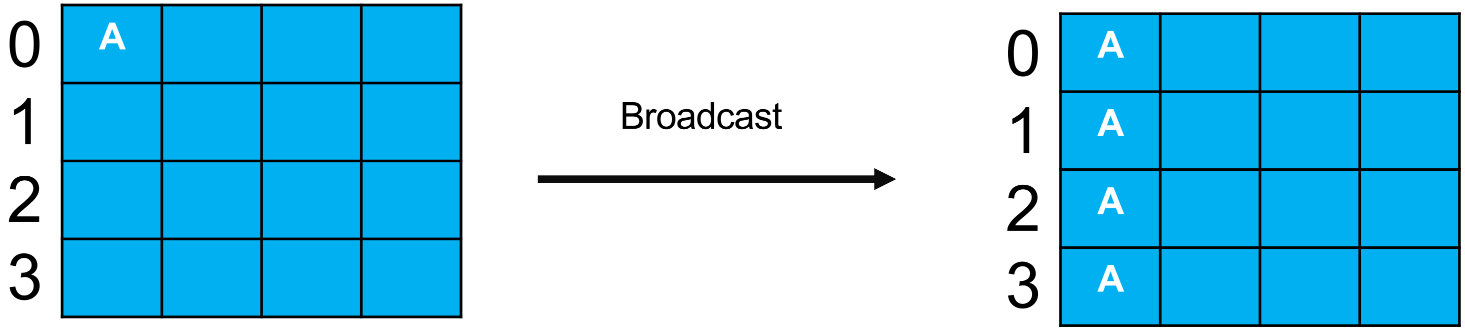
Typical collectives



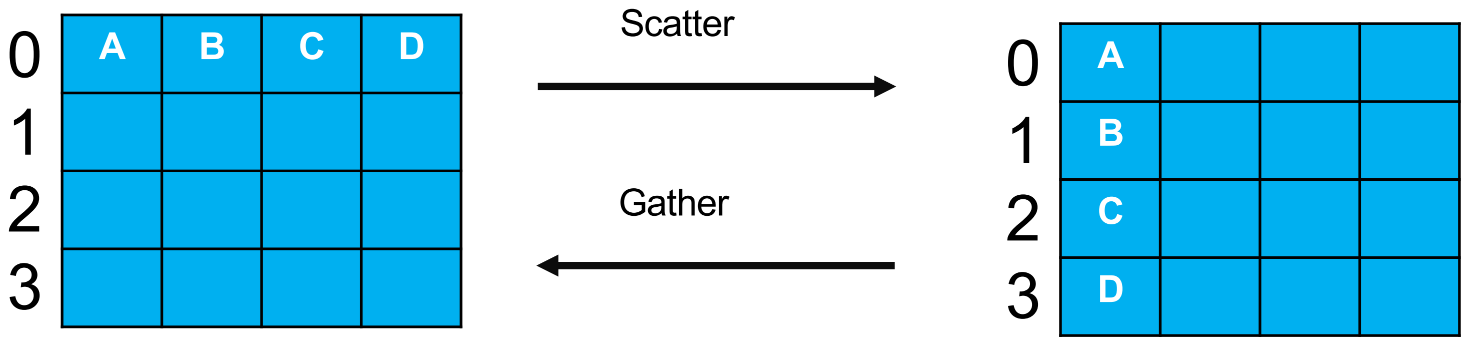
Collective data movement; Broadcast

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Example of "Rooted Collectives"



Collective data movement; Scatter, Gather



Collective data movement; Gather & Scatter

```
int MPI_Gather( const void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
     MPI_Comm comm)
```

Reverse operation

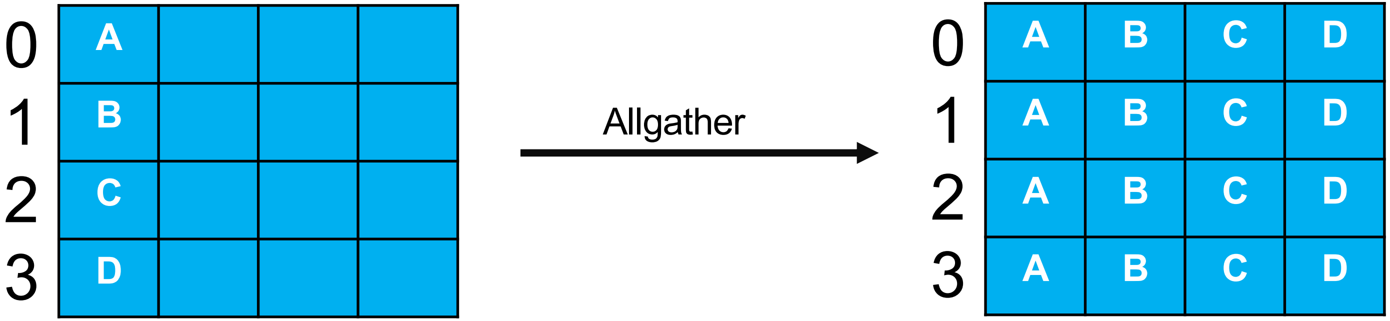
```
int MPI_Scatter (void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

- Send and receive buffers are **no longer of the same size**, hence need to specify two buffers.
- Root receives/sends np sized buffer of data, others send/receive data of the size n.
- Counterintuitively, root's **recvcount/sendcount** is **NOT** np, but n.
- SPMD code; everybody will have to allocate the large buffer; is that not awkward? Yes, other than 'root' processes,
    - use a null pointer in place of the larger buffer
    - Or use the option "MPI_IN_PLACE" for the unnecessary buffers.

Collective data movement; Allgather

```
int MPI_Allgather (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Iallgather (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm,
    MPI_Request *request)
```



Questions: What is this equivalent of (using simpler functions)?
Can this be useful in matrix ops? Which implementation is faster?

Collective data movement; Alltoall

int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm,
MPI_Request *request)

0	A0	A1	A2	A3
1	B0	B1	B2	B3
2	C0	C1	C2	C3
3	D0	D1	D2	D3

$\xrightarrow{\text{Alltoall}}$

0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

Questions: What is this operation actually doing? Can it be useful in matrix manipulations?

Special variants

- The basic routines send/receive the same amount of data from each process
- "v" for vector routines to allow the programmer to specify a message of different length for each destination (one-to-all) or source (all-to-one) or destination and source (all-to-all)

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_Alltoallv (void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype,
        void *recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

- May need to use some other collectives to compute the required displacements

MPI/Coll_1.c

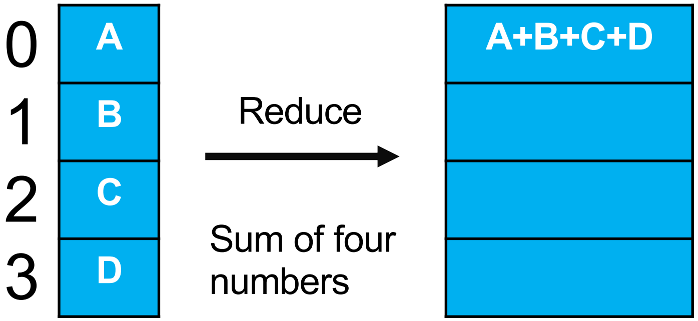
Collective computation

- **Combines communication with computation**
- **Combination operations either**
  - **Predefined**
  - **User defined**

Pre-defined

MPI type	meaning	applies to\
MPI_MAX	maximum	integer, floating point
MPI_MIN	minimum	
MPI_SUM	sum	integer, floating point, complex, multilanguage types
MPI_REPLACE	overwrite	
MPI_NO_OP	no change	
MPI_PROD	product	
MPI_LAND	logical and	C integer, logical
MPI_LOR	logical or	
MPI_LXOR	logical xor	
MPI_BAND	bitwise and	integer, byte, multilanguage types
MPI_BOR	bitwise or	
MPI_BXOR	bitwise xor	
MPI_MAXLOC	max value and location	MPI_DOUBLE_INT and such
MPI_MINLOC	min value and location	

Collective computation; Reduce



Collective computation; Reduce

All-to-one, "Rooted"

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
Int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
    MPI_Request *request)
```

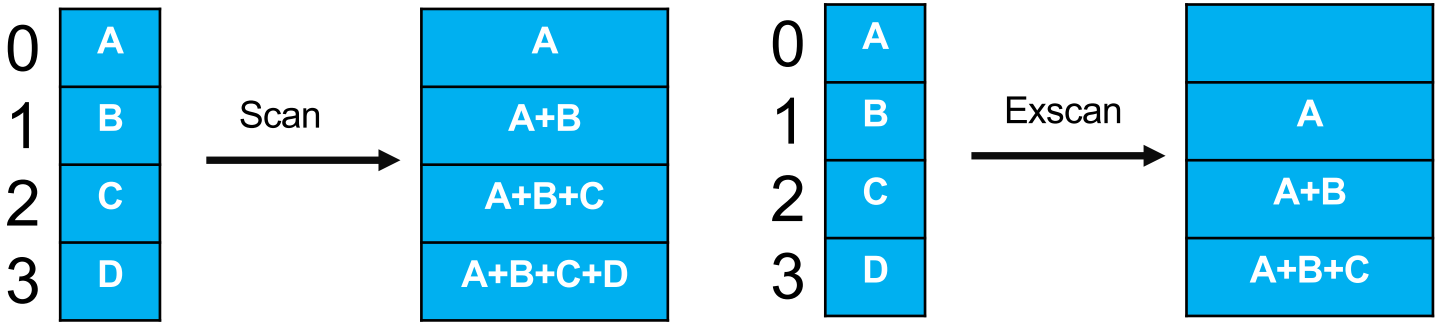
The reduction result will be returned only to root process receive buffer

All-to-all

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
    MPI_Request *request)
```

The reduction result will be returned to every rank's receive buffer.

More collective computation functions



int **MPI_Scan**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int **MPI_Iscan**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)

int **MPI_Exscan**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int **MPI_Iexscan**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)

More collective computation functions

```
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],MPI_Datatype
            datatype, MPI_Op op, MPI_Comm comm)

int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],MPI_Datatype
            datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
```

User defined operations

int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

Prototype
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
MPI_Datatype *datatype);

inoutvec[i] = invec[i] op inoutvec[i] from i=0;len-1

- The operation is assumed to be associative
- You can use flag “commute” to indicate whether the function is in addition commutative or not.
- Void return as no errors are expected

Synchronization

int MPI_Barrier(MPI_Comm comm)

int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)

- Waits until all processes have called it
- Forces time synchronization
- Not needed very often, as collectives impose synchronization on their own

About communicators

So far we have used the default communicator only

MPI_Comm comm = MPI_COMM_WORLD;

But you can do much more with them, and here we just give a short introduction to those possibilities

- Duplicate
- Split
- Define new communicators by groups of processes
- Spawn new communicators (highly advanced MPI)
- Intercommunicate (highly advanced MPI)

Duplicating

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm,
    MPI_Request *request)

Sounds strange, but is handy. Consider the following scenario

```
MPI_Isend(...);
// library call that also issues sends and receives
MPI_Irecv(...);
MPI_Waitall(...);
```


int MPI_Comm_free(MPI_Comm *comm);

Splitting

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

comm: communicator (handle)

color: control of subset assignment (integer)

key: control of rank assignment (integer)

newcomm: new communicator (handle)

Splitting: problem

How to split a 2D grid of processes into a column communicator?

MPI/Split_1.c

Constructing new by groups

- Get group of communicator

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

comm : Communicator (handle)

group : Group in communicator (handle)

- Manipulate the groups with functions like MPI_Group_incl, MPI_Group_excl, …
- Create the communicator(s) by

Int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

newcomm : new communicator (handle).

Constructing new by groups

**int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
MPI_Group *newgroup)**

group Group (handle).

n Number of elements in array ranks (and size of *newgroup*)(integer).

ranks Ranks of processes in group to appear in newgroup (array of integers).

Constructing new by groups

```
int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
        MPI_Group *newgroup)
```

group Group (handle).

n Number of elements in array ranks (integer).

ranks Array of integer ranks in group not to appear in newgroup.

Constructing new by groups: question

How to set up groups based on even or odd rank of processes?

MPI/Split_2.c

Using groups to improve one-sided comms

- Define exposure epoch, on target, and access epoch, on origin, epochs using process groups
- Target runs exposure epoch by issuing

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

- Origin runs access epoch by issuing

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

Using groups for one-sided comms: example

if (my_id==origin) {

MPI_Group_incl(all,1,&target,&tgroup);

// access

MPI_Win_start(tgroup,0,the_window);

MPI_Put(/* data on origin: */ &my_number, 1,MPI_INT, /* data on target: */ target,0, 1,MPI_INT, the_window);

MPI_Win_complete(the_window); …}

if (my_id==target) {

MPI_Group_incl(all,1,&origin,&ogroup);

// exposure

MPI_Win_post(ogroup,0,the_window);

MPI_Win_wait(the_window); …}

Intercommunicators

What if your subcommunicators would need to communicate?

Can be achieved with intercommunicators (highly advanced MPI).

Look up the function MPI_Intercomm_Create from openMPI manual

Both p2p and collective comms then possible between the subcommunicators through this intercommunicator.

User defined data types

Brief introduction

- You would like to send data of different types in one and the same message
- Your data is not contiguous

Defining and decommissioning a new datatype

```
int MPI_Type_XXX(…MPI_Datatype oldtype,
        MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *newtype)…
int MPI_Type_free(MPI_Datatype *newtype)
```

newtype the new datatype to commit, use, and decommission

oldtype the datatype to use for constructing newtype

XXX stands for one of the constructors

Datatype constructors

`MPI_Type_contiguous`	contiguous datatypes
`MPI_Type_vector`	regularly spaced datatype
`MPI_Type_indexed`	variably spaced datatype
`MPI_Type_create_subarray`	subarray within a multi-dimensional array
`MPI_Type_create_hvector`	like vector, but uses bytes for spacings
`MPI_Type_create_hindexed`	like index, but uses bytes for spacings
`MPI_Type_create_struct`	fully general datatype

Contiguous data

```
int MPI_Type_contiguous(int count,MPI_Datatype
        oldtype,  MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *newtype)…
int MPI_Type_free(MPI_Datatype *newtype)
```

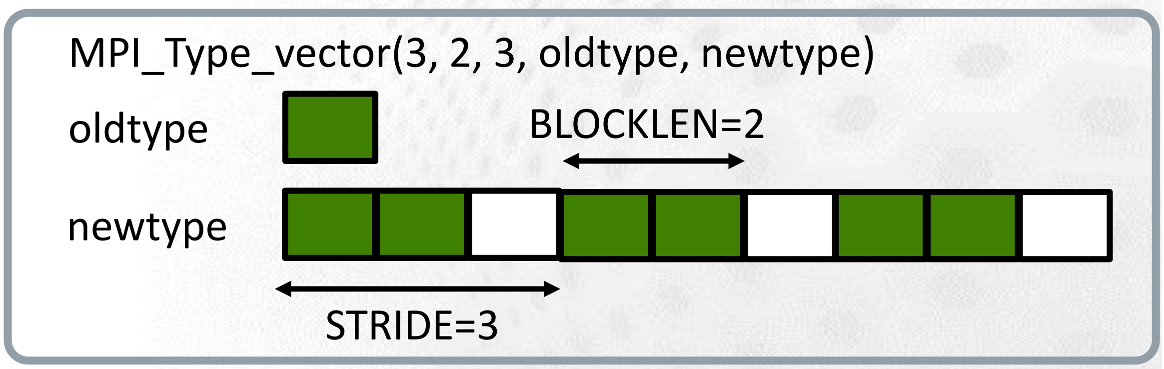
newtype the new datatype to commit, use, and decommission
oldtype the datatype to use for constructing newtype
count number of replicas

Vector data

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

count number of blocks
blocklength number of replicated oldtype elements in each block
stride total number of elements in each block

MPI/Datat_1.c



Aalto University
School of Science

Subarrays of data

int MPI_Type_create_subarray(int ndims, const int sizes[], const int subsizes[], const int offsets[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

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

Subarrays of data; simple example

```
int array_size[2] = {5,5};
int subarray_size[2] = {2,2};
int subarray_start[2] = {1,1};
MPI_Datatype subtype;
double **array
// Put in some data to the subarray of rank 1
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
```

Topologies

- **How to tell to MPI your wish to map created ranks onto the physical topology?**
- **MPI provides routines to create new communicators that order the process ranks in a way that *may* be a better match for the physical topology**
- **Virtual topologies supported**
  - Cartesian grid
  - Graph
- **Topology routines all create a *new* communicator with properties of the specified virtual topology**

Cartesian grid topology

- Useful if two neighbours in each dimension; think of a von Neumann stencil; contrast it to Moore's stencils
- Even though multi-dimensional topologies are usually the most performant, allowing MPI to use this mapping may still not give great performance gains.
- May simplify your code, though.

Cartesian grid topology

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

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords);

int MPI_Cart_rank(MPI_Comm comm, init coords, int *rank);

ndims f.ex. 2 for 2-dim, 3 for 3-dim
dims of grid in each ndim (size of ndim)
periods which of the boundaries are periodic?
reorder can MPI re-order ranks as to what it sees optimal?
coords Coordinate of the process in the Cartesian topology
rank the rank of the process in the Cartesian topology

MPI/Comm_1.c

Cartesian grid topology

Determine the neighbors for communication

int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source, int *dest)

direction Shifting direction in the defined dim
displ displacement in ranks >0 for up <0 down in the direction
source Neighbor rank in decreasing index
dest Neighbor rank towards increasing index

"Names" of the neighbor ranks come from MPI_Sendrecv, in the context of which this routine is often used

MPI/Comm_1.c

Graph topologies

- More elegant way of defining complex recurring communication patterns
- Graph vertices represent processes
- Edges denote interactions with neighbours
- Weights can be assigned to describe additional information on the edges

Graph topologies

Int MPI_Dist_graph_create_adjacent(MPI_Comm oldcomm, int indegree, int sources[], int sourceweights[], int outdegree, int dests[], int destweights[], MPI_Info info, int reorder, MPI_Comm *newcomm)

indegree : number of source nodes; **sources :** array containing the ranks of the source nodes; **sourceweights :** weights for source to destination edges or **MPI_UNWEIGHTED; outdegree :** array specifying the number of destinations, **dests :** ranks of the destination nodes, **destweights :** weights for destination to source edges or **MPI_UNWEIGHTED; info :** hints on optimization and interpretation of weights, **reorder :** the process may be reordered?

Graph topologies

int MPI_Dist_graph_create (MPI_Comm comm_old, int n, const int sources[], const int degrees[], const int destinations[], const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

n: number of source nodes; **sources :** array containing the ranks of the source nodes; **degrees :** array specifying the number of destinations for each source node, **destinations :** ranks of the destination nodes, **weights :** weights for destination to source edges or **MPI_UNWEIGHTED; info :** hints on optimization and interpretation of weights, **reorder :** the process may be reordered?

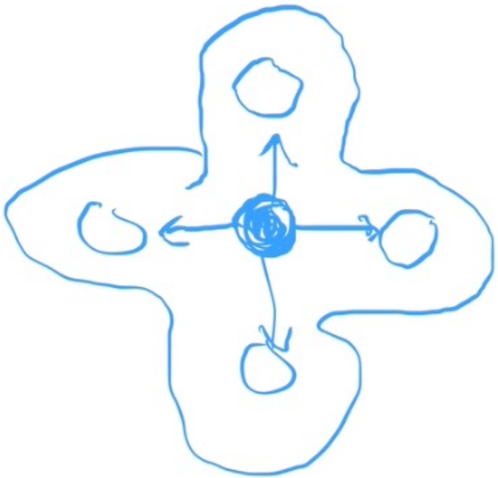
MPI_Dist_graph_neighbors &
MPI_Dist_graph_neighbors_count

Graph topologies: example

2nd order von Neumann stencil halo communication

$n = 1$

degrees $= 4$



Neighbor collectives

- **Once the graph topology has been successfully defined, then neighbor collectives, such as MPI_Neighbor_gather, MPI_Neighbor_allgather and derivatives can be used to collect data only applying to this neighborhood.**
- **This can have certain benefits over p2p communication**
  - More optimized topology
  - Collectives may use more efficients ways of communication (pipelining, trees)