Introduction to Parallel & Distributed Programming

Lec 20— MPI Collectives

Subodh Sharma | March 30, 2026

MPI DataTypes

- MPI_Datatype objects are of type:
  - Char, Short, Int, Long, LongLong Int, Float, Double, Byte, WChar
  - Signed and unsigned variants
- Derived Datatypes: User defined types over primitive types
  - MPI doesn’t understand language’s layout (struct, union, etc.)
  - Typemap: (type0, disp0), .., (typeN, dispN)
    - ith entry is of type_i and starts at byte base + disp_i

Derived DataTypes: Example

Example of MPI_Type_create_struct

```
typedef struct {
    int id;
    double value;
} Item;
```

- Lifecycle:
  - Construct
  - Commit
  - Use
  - Fee

```
/* There are 2 blocks: one int and one double */
int blocklengths[2] = {1, 1};
MPI_Aint displacements[2];
MPI_Datatype types[2] = {MPI_INT, MPI_DOUBLE};

/* Compute correct byte offsets */
MPI_Aint base;
MPI_Get_address(&x, &base);
MPI_Get_address(&x.id, &displacements[0]);
MPI_Get_address(&x.value, &displacements[1]);

displacements[0] -= base;
displacements[1] -= base;

/* Create and commit the derived datatype */
MPI_Type_create_struct(2, blocklengths, displacements, types, &item_type);
MPI_Type_commit(&item_type);
```

Derived DataTypes: Example

Example of MPI_Type_contiguous

- Creates a datatype that is just N copies of an old datatype placed back-to-back
  - More useful when the **base type itself is derived**
    - **Q: What if the base type is non-contiguous?**
    -

```
MPI_Datatype four_ints;
MPI_Type_contiguous(4, MPI_INT, &four_ints);
MPI_Type_commit(&four_ints);

/* now count = 1 means 4 ints */
MPI_Send(a, 1, four_ints, 1, 0, MPI_COMM_WORLD);

MPI_Type_free(&four_ints);
```

Derived DataTypes: Example

Example of MPI_Type_vector

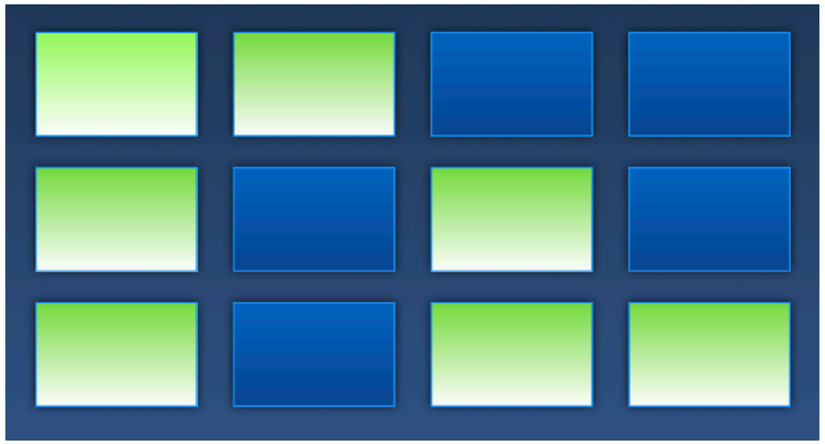
- Creates a strided regular pattern: repeated blocks of equal size, equally spaced
  - # of blocks of block length separated by stride in units of KnownType
  - In the example, what elements of A make the col_t vector?
- If want to specify stride in bytes then use: **`MPI_Type_create_hvector`**

```c
double A[4][4];
MPI_Datatype col_t;

/* 4 blocks, 1 element per block, stride 4 doubles */
MPI_Type_vector(4, 1, 4, MPI_DOUBLE, &col_t);
MPI_Type_commit(&col_t);

/* send column 2 starting at &A[0][2] */
MPI_Send(&A[0][2], 1, col_t, 1, 0, MPI_COMM_WORLD);

MPI_Type_free(&col_t);
```

Derived DataTypes: Example

Example of MPI_Type_indexed

- MPI_Type_indexed(count,
  array_of_blocklengths,
  array_of_offsets, knowntype, &newtype);
  - For the example: 5, [2,1,1,1,2], [0,4,6,8,10]
  - Blocks can contain different number of copies of the known type
  - And may have different strides
  - But the same datatype
  - Used for non-uniform gather pattern



Collective Communication

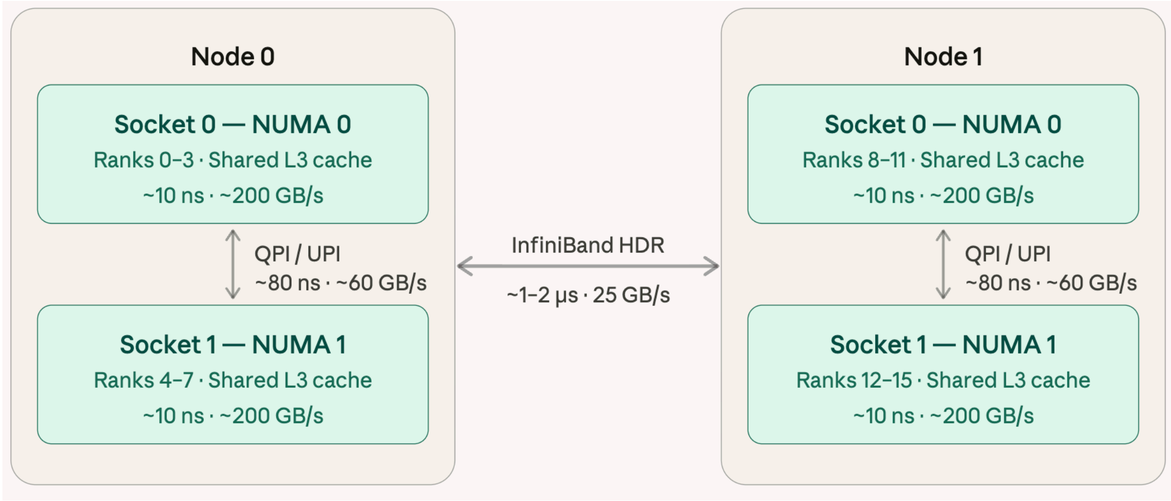
Collective Communication Primitives

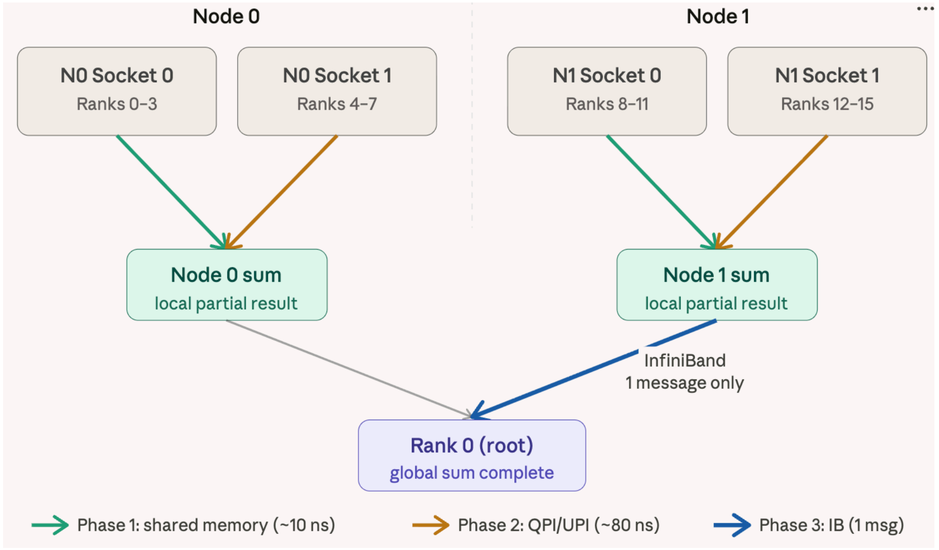
- All processes in a communicator participate
  - Are they always synchronising?
  - Do they have ordering semantics?
- Why would one choose collectives over point-to-point?
  - Helps code bloat
  - **More efficient**
    - Bcast by P-2-P sends to N procs will incur an O(N) complexity at the NIC of the root
    - But through **recursive doubling**, time complexity reduces to O(log N); better utilisation of NIC of all procs in the system

```
Round 1:  0 → 1
Round 2:  0 → 2,  1 → 3
Round 3:  0 → 4,  1 → 5,  2 → 6,  3 → 7
```

Collective Communication Primitives

- Why would one choose collectives over point-to-point?
  - Helps code bloat
  - **More efficient**
  - Knowledge of the topology of the physical interconnect
    - P-2-P will have 8 infiniband messages
    - Only one infiniband message with knowledge of topology



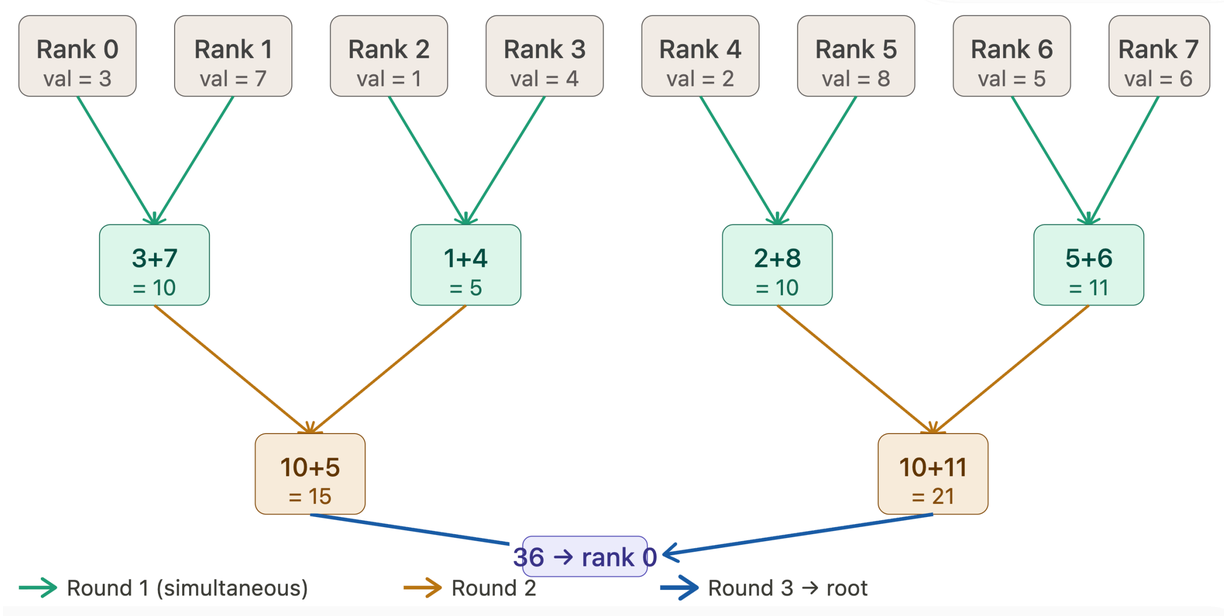


MPI Bcast

- **int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)**
  - Blocking call
  - Root and comm must match across all calls
- A nonblocking version also exists: MPI_Ibcast(…, req, comm)
  - These have to be later accompanied by an MPI_Test or an MPI_Wait

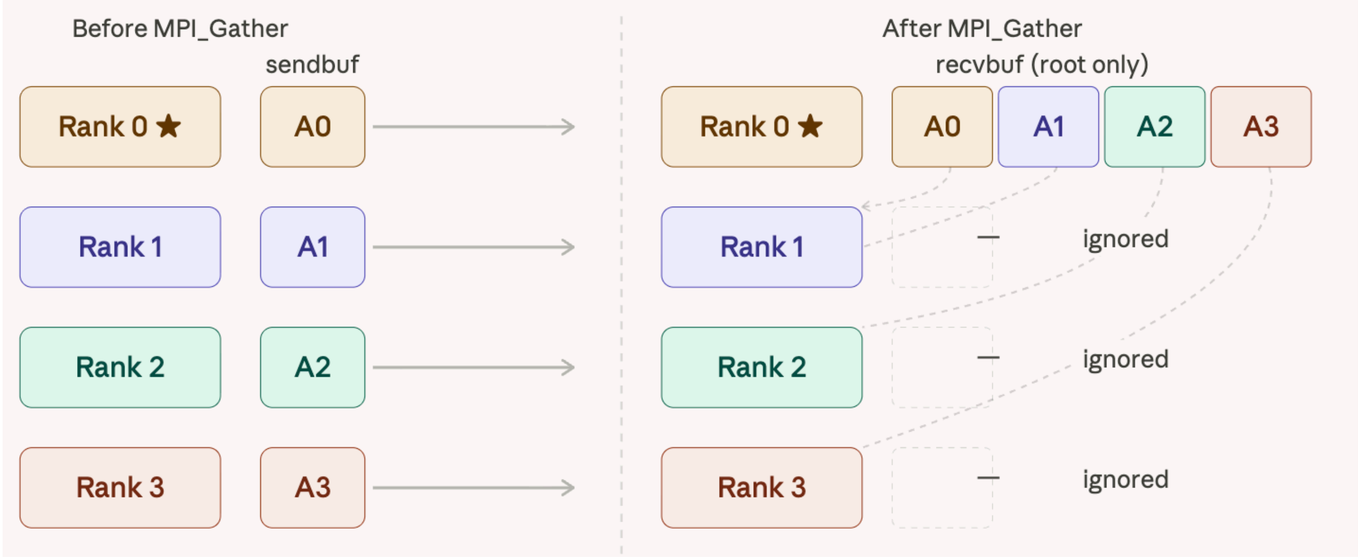
MPI_Reduce

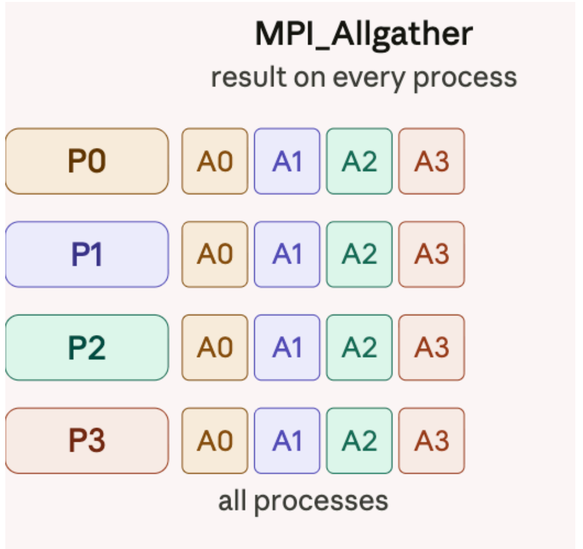
- Applies a binary operation (sum, max, logical AND, etc.) across all ranks and delivers the **single final result to one root rank**
  - MPI_Reduce(&local_val, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)
- Also check **MPI_Scan**
  - Implements prefix scan
-



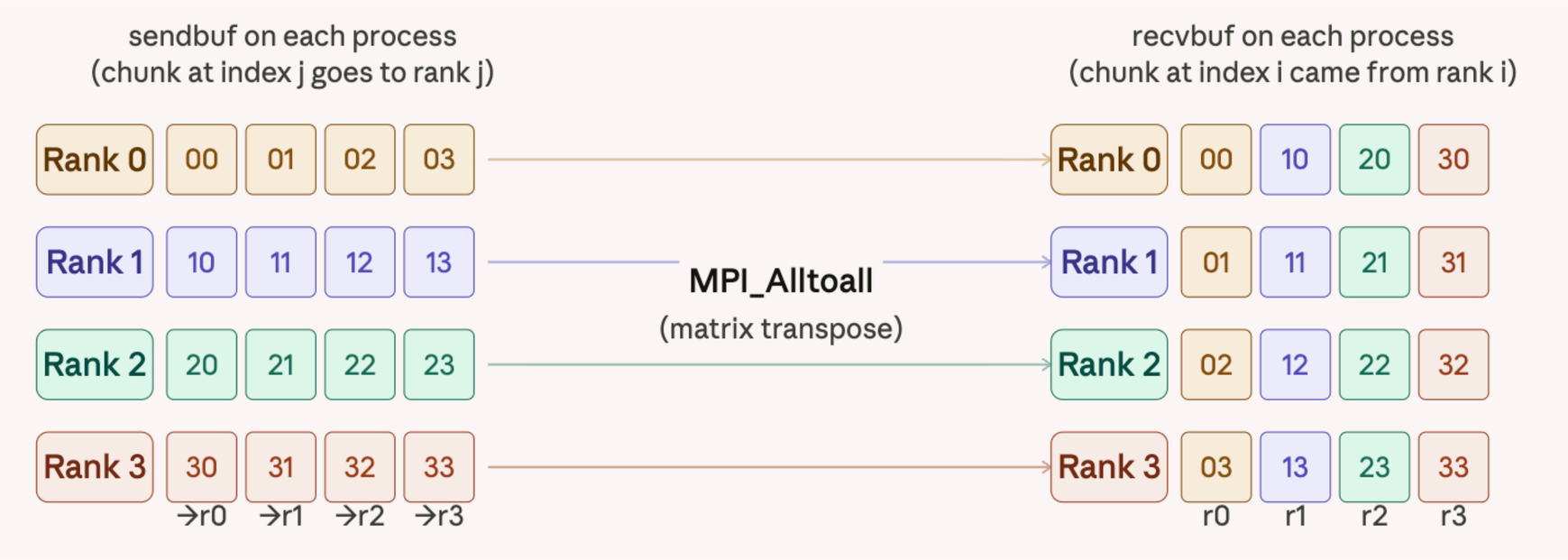
MPI_Gather

- MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);
  - Similar to non-roots sending
- MPI_Gatherv allows different size data to be gathered
- MPI_Allgather has no root; all nodes receive similarly
- Also check **MPI_Scatter**



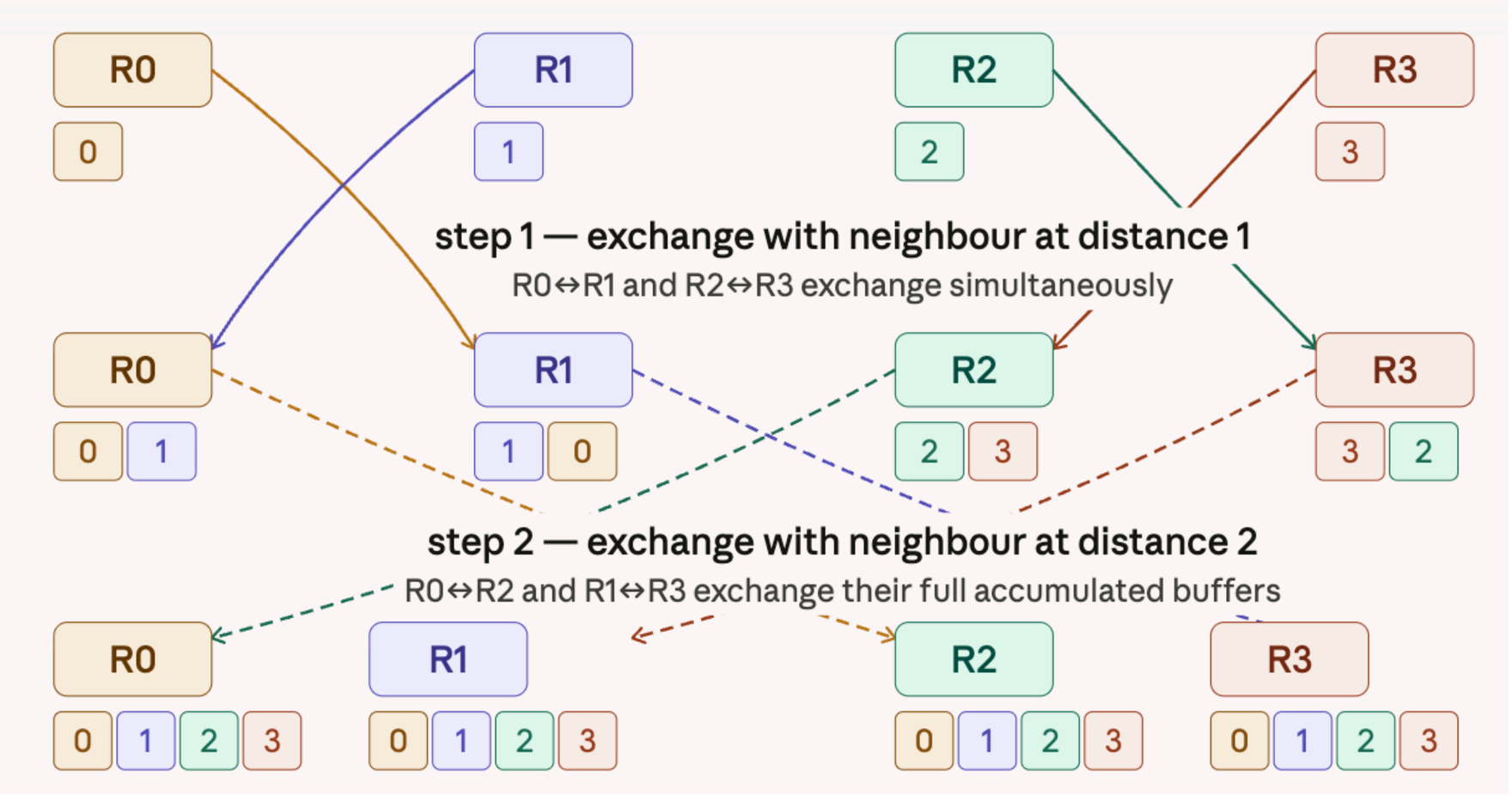


MPI_Alltoall



- MPI_Alltoall(
  - const void *sendbuf, int sendcount, MPI_Datatype sendtype,
  - void *recvbuf, int recvcount, MPI_Datatype recvtype,
  - MPI_Comm comm);
  - Naive: $O(n^2)$
  - Bruck Alg:
    - At step k: Each proc exchanges with its partner at distance 2^k
      - after $log\ n$ steps, each proc has collected n chunks

MPI_Alltoall



Communication Classification

- **One to All:** Bcast, Scatter, Scatterv
- **All to One**: Gather, Gatherv, Reduce
- **All to All**: Allgather(v), Allreduce, AlltoAll, Reduce_Scatter
- **Synchronisation**: Barrier, Scan (prefix reduction)

Other MPI Calls — Reading Assignment

One Sided Remote Memory Access

- MPI_Win_*:
  - Create, Win_fence, Win_lock/unlock, Win_flush, Win_flush_all …
- MPI_Put: writing to a remote window
- MPI_Get: reading from a remote window