MPI One-Sided Communication (RMA) - Short overview

[0] A Paradigm Shift

From Messaging to Direct Access

Standard MPI (`MPI_Send` / `MPI_Recv`) follows a **Two-Sided** communication pattern. It is like sending an email:

1. Sender writes the message.
2. Receiver must actively check their inbox and "open" it.
3. **Constraint:** If the receiver is busy doing math, the sender (or the data) waits.

One-Sided Communication (Remote Memory Access - RMA) is like a shared whiteboard:

1. Process A writes directly onto Process B's whiteboard.
2. Process B does not need to stop its work to receive the data.
3. **Benefit:** This decouples data movement from process synchronization.

The Hardware Reality: RDMA

RMA is designed to exploit **RDMA (Remote Direct Memory Access)** hardware. Modern Network Interface Cards (NICs) can read/write directly to main memory without interrupting the CPU. RMA is the software interface for this hardware capability.

---

[1] The Memory Model (Windows)

To allow remote processes to access local memory, you must explicitly "expose" a region of memory. This exposed region is called a **Window**.

1.1 The Four Window Parameters

As we have discussed int he slides, if you were designing `MPI_Win_create`, you would need four specific pieces of information to create a blueprint of the memory:

1. **Base Pointer (`void *base`):** Where does the memory start?
2. **Size (`MPI_Aint size`):** How many bytes are available? (Prevents writing off the edge).
3. **Displacement Unit (`int disp_unit`):** The unit of measure for offsets.
    - If `disp_unit = 1`: Offsets are in bytes.
    - If `disp_unit = sizeof(int)`: Offsets are in integers (indices).
4. **Communicator (`MPI_Comm`):** The group of processes allowed to participate.

1.2 The Window Object

The `MPI_Win` handle represents the **collection** of all exposed memory chunks across the communicator. Rank 0 uses this handle to "look" at Rank 1's memory.

1.3 Allocation Strategy

How you allocate the memory matters significantly for RDMA performance.

Method A: The Standard Way (`MPI_Win_create`)

You use `malloc()` yourself, then pass the pointer to MPI.

- **The Problem:** The OS treats `malloc` memory casually. It may move it physically or swap it to disk (paging).
- **The Cost:** The NIC cannot safely access moveable memory. The CPU typically has to copy data into a temporary buffer first.

Method B: The High-Performance Way (`MPI_Win_allocate`)

You ask MPI to allocate the memory *and* create the window.

- **The Solution:** MPI requests **Pinned Memory** (Page-Locked) from the OS.
- **The Benefit:** The physical address is fixed. The NIC can perform Zero-Copy RDMA transfers.

Syntax Example:

```
MPI_Win win;
int *my_data; // We declare the pointer, but don't malloc it
MPI_Aint size = 1000 * sizeof(int);

// MPI allocates memory and returns the pointer in 'my_data'
MPI_Win_allocate(size, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &my_data, &win);

// Use my_data as a normal array...
MPI_Win_free(&win); // Frees the window AND the memory
```

---

[2] Data Transfer

RMA operations are **Non-Blocking**. When a function like `MPI_Put` returns, it means the request is *queued*. It does **not** mean the data has arrived.

2.1 The Coordinate System

RMA does not use memory addresses for targets. It uses **Rank + Displacement**.
$TargetAddress = WindowBase + (TargetDisp \times DispUnit)$

2.2 The three possible actions

A. MPI_Put (Remote Write)

Concept: A remote `memcpy`.

```
MPI_Put(
    origin_addr, origin_count, origin_type,  // FROM (Me)
    target_rank, target_disp, target_count, target_type, // TO (Them)
    win
);
```

B. MPI_Get (Remote Read)

Concept: Reaching into remote memory and pulling data back.

- **Observation:** `MPI_Put` is often slightly faster than `MPI_Get` on some networks because the Origin knows exactly when the data is ready to send, whereas `Get` requires a round-trip request.

C. MPI_Accumulate (Remote Update)

Concept: Resolving the "Read-Modify-Write" race condition.
If two processes try to `Get` a counter, increment it, and `Put` it back simultaneously, updates will be lost.
`Accumulate` sends the *instruction* to the target to be performed atomically.

Operations (`MPI_Op`): `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_REPLACE` (Atomic write).

Advanced: `MPI_Get_accumulate`
Performs a "Fetch-and-Add". It updates the remote value and returns the *previous* value to the Origin. Essential for implementing shared locks or ticket counters. See details in the slides.

---

[3] Synchronization

Since data transfer is non-blocking, we need synchronization to mark **Access Epochs**.

- **Start Epoch:** "I am initiating transfers."
- **End Epoch:** "Wait until transfers are safe/complete."

3.1 Active Synchronization (Fence)

- **: the possible performance pitfalls Mechanism:** `MPI_Win_fence(0, win)`.
- **Analogy:** A "Super Barrier."
- **Behavior:** Collective call. Everyone stops.
- **Timeline:**
    1. Fence (Sync).
    2. RMA Ops (Put/Get).
    3. Fence (Wait for completion).
- **Use Case: Bulk Synchronous** applications (e.g., Stencil codes, Ghost Cell exchanges) where everyone

updates neighbors simultaneously.

3.2 Generalized Active Synchronization (PSCW)

- **Mechanism:** Post-Start-Complete-Wait.
- **Analogy:** A handshake between specific groups.
- **Concept:** Separates the **Exposure Group** (Target) from the **Access Group** (Origin).
- **Workflow:**
    1. Target calls `MPI_Win_post(group)`: "I am ready."
    2. Origin calls `MPI_Win_start(group)`: "I am accessing."
    3. Origin calls `MPI_Win_complete()`: "I am done."
    4. Target calls `MPI_Win_wait()`: "I am finished serving."
- **Use Case:** Sparse graphs or irregular topology where Rank 0 talks to Rank 1, but Rank 500 is uninvolved.

3.3 Passive Synchronization (Lock/Unlock)

- **Mechanism:** `MPI_Win_lock` / `MPI_Win_unlock`.
- **Analogy:** A library. You enter, read a book, and leave. The librarian (Target) doesn't stop working.
- **Behavior:** The Target process makes **NO** MPI calls. The Origin handles everything.
- **Lock Types:**
    - `MPI_LOCK_SHARED`: Multiple processes can read at once.
    - `MPI_LOCK_EXCLUSIVE`: Only one process can access (for writing).
- **Flush:** `MPI_Win_flush(rank, win)` ensures operations complete without releasing the lock (useful for "Write then Read" sequences).

---

[4] Code Snippets

Example A: Ghost Cell Exchange (Fence)

Scenario: 1D domain decomposition. Updating boundary cells.

```
// 1. Allocate Pinned Memory
MPI_Win_allocate(size, sizeof(int), MPI_INFO_NULL, comm, &data, &win);

// 2. Start Epoch (Open gates)
MPI_Win_fence(0, win);

// 3. Data Transfer (Push to neighbors)
// Using logic: disp_unit is sizeof(int), so offset N+1 is valid.
if (have_left_neighbor)
    MPI_Put(&data[1], 1, MPI_INT, left_rank, N+1, 1, MPI_INT, win);

if (have_right_neighbor)
    MPI_Put(&data[N], 1, MPI_INT, right_rank, 0, 1, MPI_INT, win);
```

```
// 4. End Epoch (Close gates and wait)
MPI_Win_fence(0, win);
```

Example B: Unidirectional Push (PSCW)

Scenario: Rank 0 updates Rank 1. Rank 2-999 are unaffected.

```
// Origin (Rank 0)
MPI_Win_start(group_rank_1, 0, win);
MPI_Put(..., 1, 0, ..., win); // Write to Rank 1
MPI_Win_complete(win);

// Target (Rank 1)
MPI_Win_post(group_rank_0, 0, win);
// ... Rank 1 can do local CPU work here ...
MPI_Win_wait(win); // Block until Rank 0 is done
```

Example C: Random Access (Lock)

Scenario: Rank 0 surgically reads a value from Rank 1 without Rank 1 knowing.

```
// Rank 0 (Origin)
MPI_Win_lock(MPI_LOCK_SHARED, 1, 0, win); // Lock Rank 1
MPI_Get(&val, 1, MPI_INT, 1, 0, 1, MPI_INT, win); // Request Data
MPI_Win_unlock(1, win); // Block until data arrives and release lock
```

---

[5] Summary

Topic	Best Practice / Observation
Allocation	Use `MPI_Win_allocate` whenever possible for **pinned memory** (RDMA) speed.
Displacement	Set `disp_unit` to `sizeof(datatype)` to simplify index math.
Put vs Get	`Put` is conceptually a "remote write". `Get` is a "remote read".
Atomicity	Never use `Put` + `Get` to update shared counters. Use `MPI_Accumulate`.
Fence	Use for **Bulk Synchronous** (easy, collective, stops everyone).
Lock	Use for **Asynchronous/Sparse** (fast, one-sided, requires care).
Flush	Use within a Lock epoch to sync data without losing the lock.